Títol: **"Disseny i implementació d'un processador en un llenguatge de descripció de hardware"**

Volum: 1

Alumne: Marta Miralpeix Anglerill

Director: Jordi Cortadella Fortuny
Codirector: Josep Carmona Vargas

Departament: Llenguatges i Sistemes Informàtics

Data: Març 2011

DADES DEL PROJECTE

Títol del Projecte: Disseny i implementació d'un processador en un llenguatge de descripció de Hardware.

Nom de l'estudiant: Marta Miralpeix Anglerill
Titulació: Enginyeria Informàtica Superior
Crèdits: 37,5
Director: Jordi Cortadella Fortuny
Codirector: Josep Carmona Vargas
Departament: Llenguatges i Sistemes Infomàtics

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President: Marc Solé Simó

Vocal: Fernando Martínez Sáez

Secretari: Jordi Cortadella Fortuny

QUALIFICACIÓ

Qualificació numèrica:

Qualificació descriptiva:

Data:

ÍNDEX

Disseny i Implementació d'un processador en un llenguatge de descripció de hardware

Memòria Descriptiva

INTRODUCCIÓ

El capítol de Memòria Descriptiva té com a objectiu presentar els aspectes menys tècnics del projecte **"Disseny i implementació d'un processador en un llenguatge de descripció de hardware"**. Els següents apartats descriuen, organitzen, planifiquen, i exposen les conclusions d'aquest projecte realitzat per al departament de Llenguatges i Sistemes Informàtics de la Facultat d'Informàtica de Barcelona.

Com a descripció inicial del projecte es van proposar tres objectius bàsics, a partir dels quals es va anar evolucionant, aquest són:

- Dissenyar un processador senzill que compleixi amb els requisits de modularitat i extensibilitat.
- Implementar el processador en Verilog[1], un llenguatge de descripció de hardware (HDL).
- Programació d'una Field-Programmable-Gate-Array (FPGA)[2] amb el processador i la l'execució de programes corresponent.

El projecte inicialment plantejat tenia com a objectiu principal i transversal als tres anteriors l'ensenyament de programació de circuits amb eines HDL en un entorn docent, on es demanava la proposta d'exemples d'ús, ampliació o modificació del processador.

El projecte resultant consta del disseny i la implementació d'un processador anomenat *CAL16*[3], el qual respon a l'execució d'un llenguatge, el *CAL_assembler*[4], a més de la implementació d'un compilador[5] que reconeix aquest llenguatge. Les característiques tècniques d'aquests, es poden trobar dins el capítol Memòria Tècnica d'aquest document. En el capítol anomenat Documents Adjunts, es pot trobar un manual d'introducció al llenguatge Verilog que permet iniciar-se a la programació en Verilog, un manual d'usuari que descriu les funcionalitats de les eines per a l'execució del processador i també una col·lecció d'exercicis.

Així doncs, tal i com ja es presenta en el primer paràgraf, durant els següents apartats d'aquest capítol, es pot trobar tota la informació que fa referència al gestió del projecte, des de la seva definició detallada, fins a les conclusions finals.

---

[1] *Verilog: Verilog és un llenguatge de descripció de hardware, per a més informació veure capítol Documents Adjunts, document Introducció al Verilog.*

[2] *FPGA: Field Programmable Gate Array. Dispositiu de lògica programable que pot reproduir funcions senzilles.Concretament en aquest projecte s'utilitza la FPGA Spartan 3E.*

[3] *CAL16: Processador de 16 bits i tipus RISC que es construeix en aquest projecte. Per informació més extensa veure capítol Memòria Tècnica, apartat Processador CAL16.*

[4] *CAL_assembler: Llenguatge assemblador reconegut pel processador CAL16. Per a més informació veure capítol Memòria Tècnica, apartat CAL_assembler o apartat Annexes, Annex 4, on està definit el conjunt d'instruccions que el formen.*

[5] *Compilador: Procés que revisa la correctesa d'un programa escrit en un llenguatge de programació i el tradueix cap a un altre llenguatge. Concretament en el cas d'aquest projecte, llegeix i comprova un programa en CAL_assembler i el tradueix a binari per carregar el programa a la Memòria d'Instruccions del processador CAL16.*

METODOLOGIA

Per a la gestió del projecte s'ha decidit agafar com a model una metodologia àgil com és SCRUM[1]. S'ha aplicat una simplificació d'aquesta metodologia, ja que es tracta d'un projecte de mitja duració, en el que l'equip de treball està format per una única persona i SCRUM defineix un seguit de reunions de seguiment amb l'equip de treball que en aquest cas manquen de sentit.

El que aporta SCRUM com a metodologia, i la part que s'ha aplicat sobre la gestió del projecte és que permet una organització eficient del temps i dóna la possibilitat de introduir canvis de forma evolutiva, és a dir, es seguirà un procés iteratiu de curt termini on les mateixes activitats seran exposades sota control i canvis de forma cíclica, seguint el següent esquema:

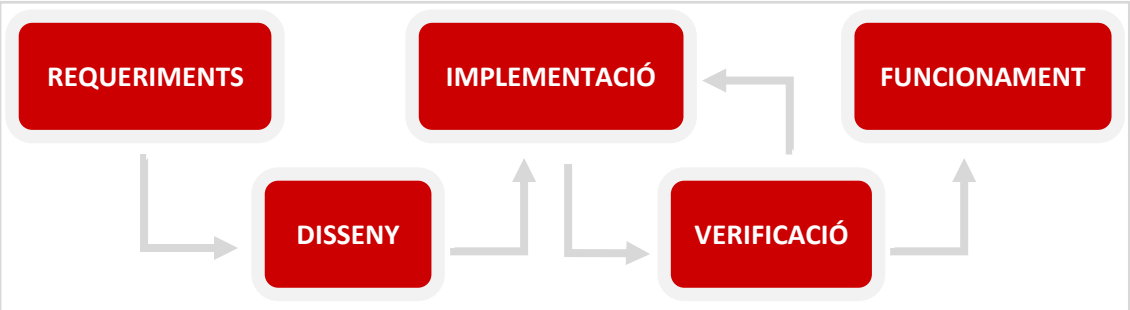


FIGURA 1 – Activitats i Procés d'evolució de la metodologia SCRUM

En cadascuna d'aquestes etapes es porta a terme:

- **Requeriments:** Etapa en la que es porta a terme l'anàlisi i la definició de requisits del projecte, en aquesta s'especifiquen els objectius i les principals funcionalitats d'aquest. L'especificació resultant es manté fins el final del projecte. D'aquesta etapa en surt la definició de quatre grans blocs sobre els que treballar:
    - El processador *CAL16* amb el seu llenguatge assemblador.
    - El compilador del llenguatge *CAL_assembler*.
    - La integració de les dues parts anteriors i l'execució de programes a nivell de simulació i mitjançant la FPGA.
    - Aplicar un enfocament docent al projecte, que és el seu objectiu principal.
- **Disseny:** Procés en el que s'especifica i es dissenya cadascuna de les parts i subparts que formen el projecte, en aquest moment s'especifica l'entorn de treball, les eines i es defineixen els requisits de software i hardware del projecte. Això ens ajuda a garantir el compliment de modularitat que es demana com a requisit principal del projecte.
- **Implementació:** En l'etapa anterior s'han dissenyat subparts per cadascuna de les parts que formen el total del projecte. En aquesta etapa s'implementa cadascuna de forma independent, i al mateix moment es va provant el correcte funcionament de cada part. A

[1] *SCRUM: Metodologia àgil utilitzada en la gestió de projecte. Per a més informació veure capítol Annexes, Annex 1.*

mesura que es tenen unitats més grans finalitzades es passa a provar les subparts en conjunt per assegurar-se que es compleixen les especificacions. D'aquesta manera un canvi imprevist queda encapsulat dins la seva unitat.

- **Integració:** Un cop implementat el *CAL16* i el compilador del llenguatge *CAL_assembler* que permet la generació de programes, la integració d'aquestes parts permet realitzar proves a nivell de funcionament global, on podem executar els programes creats, veure el seu correcte funcionament i valorar el compliment dels objectiu inicials.
- **Funcionament:** Finalment un cop el processador ja funciona, aprofitant les seves funcions, és el moment de donar-li aquest enfocament docent on es poden descobrir canvis o definir noves funcionalitats que impliquin una millora del projecte.

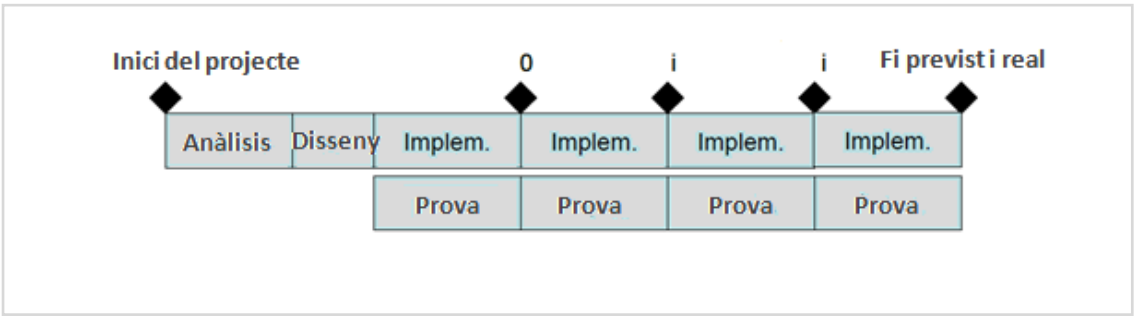


FIGURA 2 – Evolució temporal d'un projecte que utilitza la metodologia SCRUM.

OBJECTIUS

Aquest projecte busca complir quatre objectius troncals, a partir dels quals se'n poden definir d'altres més secundaris que els formen. Però alhora, té un objectiu principal respecte els altres, que és el que guia i justifica totes les decisions preses durant la seva implementació.

OBJECTIU PRINCIPAL

Es pot definir un objectiu transversal a tots els punts que inclou el projecte. Aquest objectiu és aconseguir preparar un entorn docent amb la finalitat d'ensenyar un llenguatge de la família HDL[1], com és el Verilog, mitjançant el disseny i implementació d'un processador i el seu posterior mapeig a una FPGA.

OBJECTIUS TRONCALS

Aquest propòsit és el fil conductor del projecte, del qual es deriven quatre objectius més concrets:

- Disseny del processador *CAL16*, sempre complint les característiques de modularitat i extensibilitat, fet que permetrà en un futur modificacions i ampliacions de la màquina, ja sigui per afegir nous objectius docents, com per la comprensió del processador per part dels alumnes.
- Implementació del processador en un llenguatge de descripció de hardware (HDL), com és el Verilog. Aquest objectiu ens permet disposar de l'anterior disseny físicament, i mitjançant les eines de desenvolupament de Xilinx[2], no tant sols es podrà aprendre el llenguatge de Verilog sinó que també es podrà posar en pràctica qualsevol modificació del processador i simular el seu comportament.
- Programar una Field-Programmable-Gate-Array (FPGA) amb el processador. Gràcies a aquest, es dóna al projecte un aire més pràctic, ja que es permet executar-lo amb la placa FPGA i preparar unes pràctiques de laboratori encarades als alumnes, on se'ls introdueixin aquests dispositius i les seves possibles aplicacions.
- Preparar un manual d'usuari que permeti el coneixement del processador dissenyat i un conjunt reduït de problemes i pràctiques que il·lustri amb diferents exemples possibles usos, modificacions o ampliacions d'aquest, ja sigui a nivell de simulació amb el simulador de Xilinx, com d'execució amb la FPGA Spartan 3E.

[1] ***HDL:*** *Hardware Description Language. Llenguatge de descripció de hardware que permet programar les interconnexions i el comportament d'un circuit electrònic.*

[2] ***Eines de Xilinx:*** *Entorn de desenvolupament i execució de projectes escrits en llenguatges de la família HDL.*

OBJECTIUS CONCRETS

Disseny del processador

- Disseny del processador *CAL16*, format per una Memòria de Dades, una Memòria d'Instruccions, un Espai d'Entrada/Sortida i una Unitat Central de Procés, que inclou la Unitat de Control i la Unitat de Procés.
- Disseny del llenguatge assemblador *CAL_assembler*, reconegut per la Unitat de Control i que permet executar programes en el processador.
- Disseny d'un compilador que genera un fitxer binari a partir d'un programa escrit en *CAL_assembler*.

Implementació del processador

- Implementació del processador en Verilog, del qual la compilació i la simulació es realitzen mitjançant la ISE Design Suite de Xilinx.
- Implementació del compilador que a partir d'un seguit d'instruccions que formen part del llenguatge assemblador descrit, detecti els errors o en cas de que no n'existeixin generi un fitxer binari que es podrà carregar en la memòria d'instruccions.
- Simulació de programes prèviament compilats amb el simulador de Xilinx ISE Simulator.

Mapeig del processador en una FPGA

- Aconseguir un codi sintetitzable, que es pugui programar en la FPGA.
- Mapeig del processador a la FPGA i execució d'aquest.
- Acoblament entre l'espai d'entrada del processador amb els components interruptors i de l'espai de sortida amb els leds de la FPGA.
- A partir de programes prèviament compilats, execució d'aquests amb la FPGA utilitzant els leds o els interruptors per a l'entrada i la sortida d'informació del programa.

Col·lecció d'Exercicis i Pràctiques

Desenvolupar un conjunt reduït de possibles exercicis i pràctiques amb l'objectiu de:

- Conèixer el Processador.
- Conèixer el Llenguatge Assemblador.
- Simular programes amb el processador.
- Modificar o ampliar el processador.
- Conèixer com programar una FPGA.
- Executar programes amb la FPGA Spartan 3E utilitzant els dispositius d'entrada sortida dels interruptors i els leds.

TECNOLOGIES

Aquest apartat té com a objectiu descriure aquells aspectes tecnològics amb els que s'ha desenvolupat i sobre els que s'executa el processador *CAL16*. Les característiques tecnològiques del projecte són les següents:

- El processador es desenvolupa i s'executa sota la plataforma de Windows [1]. Tot i que les eines que utilitza poden instal·lar-se també en Linux[2].
- El llenguatge de programació del processador és Verilog.
- L'entorn de desenvolupament i d'execució del processador es realitza mitjançant les eines que proporciona Xilinx ISE Design Suite 12.1, que permeten la generació, simulació, sintetització i posterior programació en una FPGA, de projectes descrits en llenguatges HDL.
- El llenguatge de programació del compilador és C++[3].
- El compilador utilitza l'eina ANTLR[4] pel reconeixement i la representació del llenguatge.
- La FPGA utilitzada és una Spartan 3E.

FIGURA 3 – FPGA Spartan 3E

[1] *Windows: Sistema operatiu desenvolupat per Microsoft.*
[2] *Linux: Sistema operatiu lliure de tipus UNIX.*
[3] *C++: Extensió del llenguatge de programació C que permet la manipulació d'estructures de dades.*
[4] *ANTLR: ANother Tool for Language Recognition. Eina de reconeixement de llenguatges a partir de descripcions gramaticals.*

CALENDARI

Per a portar a terme la planificació temporal, s'ha utilitzat el programa Microsoft Project 2007[1]. Mitjançant aquesta eina s'ha creat un projecte anomenat planificacio.mp[2] que inclou el conjunt de tasques a realitzar, aquestes s'han englobat en diferents grups per a estructurar la feina i permetre treballar de forma evolutiva en blocs de treball, sempre seguint la metodologia de treball SCRUM[3].

Aquest projecte comença el dia 1 de Juliol de 2010 i té prevista la seva finalització el 1 de Març de 2011. A la següent figura es mostra el conjunt de tasques principals que han derivat dels objectius desenvolupats en el primer apartat. S'han distribuït les tasques en grups segons l'objectiu al que pertanyen, i per cadascuna podem veure la durada (dies) i les dates d'inici i de fi.

	Nombre de tarea	Duración	Comienzo	Fin
0	⊟ **Disseny i implementació d'un processador en llenguatge de descripció de hardware**	**185 días**	**lun 14/06/10**	**lun 28/02/11**
1	⊞ **Previs**	**49 días**	**lun 14/06/10**	**jue 19/08/10**
13				
14	⊞ **Disseny i implementació del processador**	**91 días**	**lun 12/07/10**	**lun 15/11/10**
80				
81	⊞ **Llenguatge assemblador**	**105 días**	**lun 12/07/10**	**vie 03/12/10**
102				
103	⊞ **Mapeig a la FPGA**	**100 días**	**lun 06/09/10**	**vie 21/01/11**
114				
115	⊞ **Test del total**	**20 días**	**lun 24/01/11**	**vie 18/02/11**
117				
118	⊞ **Col·leccio de problemes**	**12 días**	**lun 24/01/11**	**mar 08/02/11**
135				
136	⊞ **Documents**	**87 días**	**lun 18/10/10**	**mar 15/02/11**
162				
163	⊞ **Tràmits**	**184 días**	**lun 14/06/10**	**lun 28/02/11**
170				
171	⊞ **Presentació**	**19 días**	**mar 01/02/11**	**lun 28/02/11**

FIGURA 4 – Llistat de tasques principals en les que es divideix el projecte

A continuació les següent figures mostren en detall les subtasques en les que es divideix cadascuna de les tasques principals presentades en la imatge anterior, això permet veure com s'ha estructurat cada part del projecte i la càrrega de treball que ha significat. Per a cada subtasca també es mostra el seu nom, la duració en dies, les dates d'inici i fi en portar-les a terme i una referència a aquelles tasques que necessariament la precedeixen.

[1] *Microsoft Project 2007: Software d'administració de projectes, dissenyat, desenvolupat i comercialitzat per Microsoft.*

[2] *Planificació.mp: Projecte de Microsoft Project que inclou la planificació temporal del projecte. Per a obtenir el fitxer amb la planificació temporal del projecte veure capítol Annexes, Annex 2.*

[3] *SCRUM: Metodologia àgil utilitzada en la gestió de projecte. Per a més informació veure capítol Annexes, Annex 1.*

	Nombre de tarea	Duración	Comienzo	Fin	Predec
1	**Previs**	**49 días**	**lun 14/06/10**	**jue 19/08/10**	
2	**Iniciació al Verilog**	**2 días**	**jue 01/07/10**	**vie 02/07/10**	
3	Previs - Iniciació al Verilog - Primers programes d'exemple i estudi del manual	2 días	jue 01/07/10	vie 02/07/10	
4	**Iniciació a les eines Xilinx i a la simulació**	**5 días**	**lun 05/07/10**	**vie 09/07/10**	
5	Previs - Iniciació a les eines de Xilinx i a la simulació - Estudi eines Xilinx	1 día	lun 05/07/10	lun 05/07/10	3
6	Previs - Iniciació a les eines de Xilinx i a la simulació - Exemple de SUMA_N	4 días	mar 06/07/10	vie 09/07/10	5
7	**Iniciació a la FPGA**	**20 días**	**vie 23/07/10**	**jue 19/08/10**	
8	Previs - Iniciació a la FPGA - Estudi Spartant 3E	3 días	vie 23/07/10	mar 27/07/10	
9	Previs - Iniciació a la FPGA - Exemple de SUMA_N	7 días	mié 28/07/10	jue 05/08/10	8
10	Previs - Iniciació a la FPGA - Exemple de DAU	10 días	vie 06/08/10	jue 19/08/10	9
11	Previs - Especificació d'objectius	1 día	lun 14/06/10	lun 14/06/10	
12	Previs - Planificació de les tasques	2 días	jue 01/07/10	vie 02/07/10	

FIGURA 5 – Llistat de subtasques que formen la tasca Previs

	Nombre de tarea	Duración	Comienzo	Fin	Predec
81	**Llenguatge assemblador**	**105 días**	**lun 12/07/10**	**vie 03/12/10**	
82	**Especificació**	**2 días**	**lun 12/07/10**	**mar 13/07/10**	
83	Llenguatge assemblador - Especificació - Definir sintaxi instruccions	1 día	lun 12/07/10	lun 12/07/10	
84	Llenguatge assemblador - Especificació - Definir comportament instruccions	1 día	mar 13/07/10	mar 13/07/10	83
85	**Compilador**	**30 días**	**lun 27/09/10**	**vie 05/11/10**	**82**
86	Llenguatge assemblador - Compilador - Estudi compilador CL	1 día	lun 27/09/10	lun 27/09/10	
87	Llenguatge assemblador - Compilador - Especificar compilador per fases	1 día	lun 27/09/10	lun 27/09/10	
88	**Fases**	**3 días**	**mar 28/09/10**	**jue 30/09/10**	**86;87**
89	**Anàlisis sintàctic**	**1 día**	**mar 28/09/10**	**mar 28/09/10**	
90	Llenguatge assemblador - Compilador - Fases - Anàlisis sintàctic - Implementar	1 día	mar 28/09/10	mar 28/09/10	
91	Llenguatge assemblador - Compilador - Fases - Anàlisis sintàctic - Test	1 día	mar 28/09/10	mar 28/09/10	
92	**Anàlisis semàntic**	**1 día**	**mié 29/09/10**	**mié 29/09/10**	**89**
93	Llenguatge assemblador - Compilador - Fases - Anàlisis semàntic - Implementar	1 día	mié 29/09/10	mié 29/09/10	
94	Llenguatge assemblador - Compilador - Fases - Anàlisis semàntic - Test	1 día	mié 29/09/10	mié 29/09/10	
95	**Generació de binari**	**1 día**	**jue 30/09/10**	**jue 30/09/10**	**92**
96	Llenguatge assemblador - Compilador - Fases - Generació de binari - Implementar	1 día	jue 30/09/10	jue 30/09/10	
97	Llenguatge assemblador - Compilador - Fases - Generació de binari - Test	1 día	jue 30/09/10	jue 30/09/10	
98	Llenguatge assemblador - Compilador - Test programa->compilar->carregar->simular	2 días	vie 01/10/10	lun 04/10/10	88
99	Llenguatge assemblador - Compilador - Adaptació a SO Windows	5 días	lun 01/11/10	vie 05/11/10	88
100	Llenguatge assemblador - Test per simulació: Programa contador	3 días	mar 16/11/10	jue 18/11/10	14;85
101	Llenguatge assemblador - Test per simulació: Programa enquesta	3 días	mié 01/12/10	vie 03/12/10	100

FIGURA 6 – Llistat de subtasques que formen la tasca Llenguatge Assemblador

	Nombre de tarea	Duración	Comienzo	Fin	Predece
103	**Mapeig a la FPGA**	**100 días**	**lun 06/09/10**	**vie 21/01/11**	**7;16;59**
104	Mapeig a la FPGA - Obtenir codi sintetitzable	3 días	lun 06/09/10	mié 08/09/10	
105	Mapeig a la FPGA - Estudi característiques de l'ús de la memòria de la FPGA	10 días	lun 06/09/10	vie 17/09/10	
106	**Modificacions del processador**	**50 días**	**lun 08/11/10**	**vie 14/01/11**	
107	Mapeig a la FPGA - Modificacions del processador - Implementar	25 días	lun 08/11/10	vie 10/12/10	
108	Mapeig a la FPGA - Modificacions del processador - Test per execució amb FPGA: Comprovar PC avança	10 días	lun 06/12/10	vie 17/12/10	
109	Mapeig a la FPGA - Modificacions del processador - Test per execució amb FPGA: Comprovar entrades/sortides	20 días	lun 20/12/10	vie 14/01/11	108
110	**Modificacions del compilador**	**50 días**	**lun 15/11/10**	**vie 21/01/11**	**85**
111	Mapeig a la FPGA - Modificacions del compilador - Implementar	1 día	lun 15/11/10	lun 15/11/10	
112	Mapeig a la FPGA - Modificacions del compilador - Test per execució amb FPGA: Programa contador	5 días	lun 17/01/11	vie 21/01/11	109
113	Mapeig a la FPGA - Modificacions del compilador - Test per execució amb FPGA: Programa enquesta	5 días	lun 17/01/11	vie 21/01/11	109

FIGURA 7 – Llistat de subtasques que formen la tasca Mapeig a la FPGA

	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
14	**Disseny i implementació del processador**	**91 días**	**lun 12/07/10**	**lun 15/11/10**	
15	Disseny i implementació del processador - Estudi CAL16	1 día	lun 12/07/10	lun 12/07/10	
16	**CPU**	**25 días**	**mar 13/07/10**	**lun 16/08/10**	
17	**UP**	**9 días**	**mar 13/07/10**	**vie 23/07/10**	
18	**ALU**	**3 días**	**mar 13/07/10**	**jue 15/07/10**	
19	Disseny i implementació del processador - CPU - UP - ALU - Disseny	1 día	mar 13/07/10	mar 13/07/10	
20	Disseny i implementació del processador - CPU - UP - ALU - Implementació	1 día	mié 14/07/10	mié 14/07/10	19
21	Disseny i implementació del processador - CPU - UP - ALU - Simulació i test	1 día	jue 15/07/10	jue 15/07/10	20
22	**BR**	**3 días**	**mar 13/07/10**	**jue 15/07/10**	
23	Disseny i implementació del processador - CPU - UP - BR - Disseny	1 día	mar 13/07/10	mar 13/07/10	
24	Disseny i implementació del processador - CPU - UP - BR - Implementació	1 día	mié 14/07/10	mié 14/07/10	23
25	Disseny i implementació del processador - CPU - UP - BR - Simulació i test	1 día	jue 15/07/10	jue 15/07/10	24
26	**PC**	**3 días**	**mar 13/07/10**	**jue 15/07/10**	
27	Disseny i implementació del processador - CPU - UP - PC - Disseny	1 día	mar 13/07/10	mar 13/07/10	
28	Disseny i implementació del processador - CPU - UP - PC - Implementació	1 día	mié 14/07/10	mié 14/07/10	27
29	Disseny i implementació del processador - CPU - UP - PC - Simulació i test	1 día	jue 15/07/10	jue 15/07/10	28
30	**ROTR**	**3 días**	**mié 14/07/10**	**vie 16/07/10**	
31	Disseny i implementació del processador - CPU - UP - ROTR - Disseny	1 día	mié 14/07/10	mié 14/07/10	
32	Disseny i implementació del processador - CPU - UP - ROTR - Implementació	1 día	jue 15/07/10	jue 15/07/10	31
33	Disseny i implementació del processador - CPU - UP - ROTR - Simulació i test	1 día	vie 16/07/10	vie 16/07/10	32
34	**OFFSET-III**	**3 días**	**mié 14/07/10**	**vie 16/07/10**	
35	Disseny i implementació del processador - CPU - UP - OFFSET-IN - Disseny	1 día	mié 14/07/10	mié 14/07/10	
36	Disseny i implementació del processador - CPU - UP - OFFSET-IN - Implementació	1 día	jue 15/07/10	jue 15/07/10	35
37	Disseny i implementació del processador - CPU - UP - OFFSET-IN - Simulació i test	1 día	vie 16/07/10	vie 16/07/10	36
38	**UP en comú**	**5 días**	**lun 19/07/10**	**vie 23/07/10**	**18;22;26;30;34**
39	Disseny i implementació del processador - CPU - UP - UP en comú - Disseny i connexió entre blocs	1 día	lun 19/07/10	lun 19/07/10	
40	Disseny i implementació del processador - CPU - UP - UP en comú - Implementació	1 día	mar 20/07/10	mar 20/07/10	39
41	Disseny i implementació del processador - CPU - UP - UP en comú - Simulació i test	3 días	mié 21/07/10	vie 23/07/10	40
42	**UC**	**9 días**	**lun 26/07/10**	**jue 05/08/10**	**17;82**
43	**Decodificació**	**3 días**	**lun 26/07/10**	**mié 28/07/10**	
44	Disseny i implementació del processador - CPU - UC - Decodificació - Disseny	1 día	lun 26/07/10	lun 26/07/10	
45	Disseny i implementació del processador - CPU - UC - Decodificació - Implementació	1 día	mar 27/07/10	mar 27/07/10	44
46	Disseny i implementació del processador - CPU - UC - Decodificació - Simulació i test	1 día	mié 28/07/10	mié 28/07/10	45
47	**Maquina d'estats**	**4 días**	**lun 26/07/10**	**jue 29/07/10**	
48	Disseny i implementació del processador - CPU - UC - Màquina d'estats - Especificació	1 día	lun 26/07/10	lun 26/07/10	
49	Disseny i implementació del processador - CPU - UC - Màquina d'estats - Implementació	1 día	mar 27/07/10	mar 27/07/10	48
50	Disseny i implementació del processador - CPU - UC - Màquina d'estats - Simulació i test	2 días	mié 28/07/10	jue 29/07/10	49
51	**UC en comú**	**5 días**	**vie 30/07/10**	**jue 05/08/10**	**43;47**
52	Disseny i implementació del processador - CPU - UC - UC en comú - Disseny i connexió entre blocs	1 día	vie 30/07/10	vie 30/07/10	
53	Disseny i implementació del processador - CPU - UC - UC en comú - Implementació	1 día	lun 02/08/10	lun 02/08/10	52
54	Disseny i implementació del processador - CPU - UC - UC en comú - Simulació i test	3 días	mar 03/08/10	jue 05/08/10	53
55	**CPU en comú**	**7 días**	**vie 06/08/10**	**lun 16/08/10**	**17;42**
56	Disseny i implementació del processador - CPU - CPU en comú - Disseny i connexió entre blocs	1 día	vie 06/08/10	vie 06/08/10	
57	Disseny i implementació del processador - CPU - CPU en comú - Implementació	1 día	lun 09/08/10	lun 09/08/10	56
58	Disseny i implementació del processador - CPU - CPU en comú - Simulació i test	5 días	mar 10/08/10	lun 16/08/10	57
59	**Memòries**	**14 días**	**mar 17/08/10**	**vie 03/09/10**	**16**
60	**Memòria d'instruccions**	**3 días**	**mar 17/08/10**	**jue 19/08/10**	
61	Disseny i implementació del processador - Memòries - Memòria d'instruccions - Especificació	1 día	mar 17/08/10	mar 17/08/10	
62	Disseny i implementació del processador - Memòries - Memòria d'instruccions - Implementació	1 día	mié 18/08/10	mié 18/08/10	61
63	Disseny i implementació del processador - Memòries - Memòria d'instruccions - Test	1 día	jue 19/08/10	jue 19/08/10	62
64	**Memòria de dades**	**3 días**	**mar 17/08/10**	**jue 19/08/10**	
65	Disseny i implementació del processador - Memòries - Memòria de dades - Especificació	1 día	mar 17/08/10	mar 17/08/10	
66	Disseny i implementació del processador - Memòries - Memòria de dades - Implementació	1 día	mié 18/08/10	mié 18/08/10	65
67	Disseny i implementació del processador - Memòries - Memòria de dades - Test	1 día	jue 19/08/10	jue 19/08/10	66
68	**Memòries en comú amb CPU**	**11 días**	**vie 20/08/10**	**vie 03/09/10**	**60;64**
69	Disseny i implementació del processador - Memòries - Memòries en comú amb CPU - Especificació	1 día	vie 20/08/10	vie 20/08/10	
70	Disseny i implementació del processador - Memòries - Memòries en comú amb CPU - Implementació	10 días	lun 23/08/10	vie 03/09/10	69
71	Disseny i implementació del processador - Memòries - Memòries en comú amb CPU - Test	10 días	lun 23/08/10	vie 03/09/10	69
72	**Espai E/S**	**3 días**	**lun 08/11/10**	**mié 10/11/10**	
73	Disseny i implementació del processador - Espai E/S - Especificació	1 día	lun 08/11/10	lun 08/11/10	
74	Disseny i implementació del processador - Espai E/S - Implementació	1 día	mar 09/11/10	mar 09/11/10	73
75	Disseny i implementació del processador - Espai E/S - Test	1 día	mié 10/11/10	mié 10/11/10	74
76	**Processador en comú**	**3 días**	**jue 11/11/10**	**lun 15/11/10**	**16;59;72**
77	Disseny i implementació del processador - Processador en comú - Dissenyar connexió entre blocs	1 día	jue 11/11/10	jue 11/11/10	
78	Disseny i implementació del processador - Processador en comú - Implementar	1 día	vie 12/11/10	vie 12/11/10	77
79	Disseny i implementació del processador - Processador en comú - Testeig per simulació	1 día	lun 15/11/10	lun 15/11/10	78

FIGURA 8 – Llistat de subtasques que formen la tasca Disseny i Implementació del Processador

	Nombre de tarea	Duración	Comienzo	Fin	Predec
115	**Test del total**	**20 días**	**lun 24/01/11**	**vie 18/02/11**	**103**
116	Test el total - Diferents programes->compilar->executar amb FPGA	20 días	lun 24/01/11	vie 18/02/11	

FIGURA 9 – Llistat de subtasques que formen la tasca Test total

	Nombre de tarea	Duración	Comienzo	Fin	Pred
118	**Col·leccio de problemes**	**12 días**	**lun 24/01/11**	**mar 08/02/11**	
119	**Programació al Verilog**	**3 días**	**lun 24/01/11**	**mié 26/01/11**	**113**
120	Col·lecció de problemes - Programació al Verilog - Especificar objectius	1 día	lun 24/01/11	lun 24/01/11	
121	Col·lecció de problemes - Programació al Verilog - Redactar problema per objectiu	1 día	mar 25/01/11	mar 25/01/11	120
122	Col·lecció de problemes - Programació al Verilog - Solucionar problemes	1 día	mié 26/01/11	mié 26/01/11	121
123	**Simulació de programes en CAL_assembler**	**3 días**	**jue 27/01/11**	**lun 31/01/11**	**119**
124	Col·lecció de problemes - Simulació de programes en CAL_assembler - Especificar objectius	1 día	jue 27/01/11	jue 27/01/11	
125	Col·lecció de problemes - Simulació de programes en CAL_assembler - Redactar problema per objectiu	1 día	vie 28/01/11	vie 28/01/11	124
126	Col·lecció de problemes - Simulació de programes en CAL_assembler - Solucionar problemes	1 día	lun 31/01/11	lun 31/01/11	125
127	**Execució de programes amb FPGA**	**3 días**	**mar 01/02/11**	**jue 03/02/11**	**123**
128	Col·lecció de problemes - Execució de programes amb FPGA - Especificar objectius	1 día	mar 01/02/11	mar 01/02/11	
129	Col·lecció de problemes - Execució de programes amb FPGA - Redactar problema per objectiu	1 día	mié 02/02/11	mié 02/02/11	128
130	Col·lecció de problemes - Execució de programes amb FPGA - Solucionar problemes	1 día	jue 03/02/11	jue 03/02/11	129
131	**Ampliació i modificació del processador**	**3 días**	**vie 04/02/11**	**mar 08/02/11**	**127**
132	Col·lecció de problemes - Ampliació i modificació del processador - Especificar objectius	1 día	vie 04/02/11	vie 04/02/11	
133	Col·lecció de problemes - Ampliació i modificació del processador - Redactar problema per objectiu	1 día	lun 07/02/11	lun 07/02/11	132
134	Col·lecció de problemes - Ampliació i modificació del processador - Solucionar problemes	1 día	mar 08/02/11	mar 08/02/11	133

FIGURA 10 – Llistat de subtasques que formen la tasca Col·lecció de Problemes

	Nombre de tarea	Duración	Comienzo	Fin	Predece
136	**Documents**	**87 días**	**lun 18/10/10**	**mar 15/02/11**	
137	**Informe previ**	**5 días**	**lun 15/11/10**	**vie 19/11/10**	**11;12**
138	Documents - Informe previ - Re-especificar objectius	1 día	lun 15/11/10	lun 15/11/10	
139	Documents - Informe previ - Re-planificació	2 días	mar 16/11/10	mié 17/11/10	138
140	Documents - Informe previ - Redactar informe	2 días	jue 18/11/10	vie 19/11/10	139
141	**Memòria**	**87 días**	**lun 18/10/10**	**mar 15/02/11**	
142	Documents - Memòria - Estudi de memòries de PFC's	2 días	lun 18/10/10	mar 19/10/10	
143	Documents - Memòria - Definir format, contingut i ordre memòria	1 día	mié 20/10/10	mié 20/10/10	142
144	**Redactar**	**69 días**	**lun 08/11/10**	**jue 10/02/11**	
145	Documents - Memòria - Redactar - Memòria Descriptiva	5 días	lun 17/01/11	vie 21/01/11	
146	Documents - Memòria - Redactar - Disseny del processador	4 días	lun 13/12/10	jue 16/12/10	14
147	Documents - Memòria - Redactar - Especificació del llenguatge assemblador	2 días	lun 08/11/10	mar 09/11/10	82
148	Documents - Memòria - Redactar - Compilador	4 días	lun 08/11/10	jue 11/11/10	85
149	Documents - Memòria - Redactar - Implementació del processador	5 días	lun 24/01/11	vie 28/01/11	14;81;1(
150	Documents - Memòria - Redactar - Exemples de simulació del processador	1 día	lun 13/12/10	lun 13/12/10	14;81
151	Documents - Memòria - Redactar - Característiques de la FPGA	4 días	lun 31/01/11	jue 03/02/11	7;105;1
152	Documents - Memòria - Redactar - Mapeig del processador en la FPGA	5 días	lun 24/01/11	vie 28/01/11	103
153	Documents - Memòria - Redactar - Exemples d'execució en la FPGA	5 días	lun 24/01/11	vie 28/01/11	103
154	Documents - Memòria - Redactar - Col·lecció d'exercicis	2 días	mié 09/02/11	jue 10/02/11	118
155	Documents - Memòria - Redactar - Manual Usuari	5 días	lun 24/01/11	vie 28/01/11	103
156	Documents - Memòria - Redactar - Introducció al Verilog	5 días	lun 31/01/11	vie 04/02/11	155
157	Documents - Memòria - Redactar - Posta en comú d'annexes	2 días	lun 07/02/11	mar 08/02/11	146;147
158	Documents - Memòria - Redactar - Posta en comú apartats	1 día	mié 09/02/11	mié 09/02/11	157
159	Documents - Memòria - Redactar - Posta en comú de bibliografia	1 día	jue 10/02/11	jue 10/02/11	158
160	Documents - Memòria - Reordenació de la memòria i redacció de connectors	2 días	vie 11/02/11	lun 14/02/11	159
161	Documents - Memòria - Índex i portada	1 día	mar 15/02/11	mar 15/02/11	160

FIGURA 12 – Llistat de subtasques que formen la tasca Documents

	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
163	**Tràmits**	**184 días**	**lun 14/06/10**	**lun 28/02/11**	
164	Tràmits - Inscripció	0 días	lun 14/06/10	lun 14/06/10	11
165	Tràmits - Informe previ	0 días	mar 30/11/10	mar 30/11/10	137
166	Tràmits - Matrícula	0 días	lun 07/02/11	lun 07/02/11	
167	Tràmits - Reserva data i lloc lectura	0 días	lun 07/02/11	lun 07/02/11	166
168	Tràmits - Presentar memòria	0 días	lun 14/02/11	lun 14/02/11	
169	Tràmits - Lectura	0 días	lun 28/02/11	lun 28/02/11	165FC+60 días

FIGURA 13 – Llistat de subtasques que formen la tasca Tràmits

	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
171	**Presentació**	**19 días**	**mar 01/02/11**	**lun 28/02/11**	
172	Presentació - Definir contingut presentació	1 día	mar 01/02/11	mar 01/02/11	
173	Presentació - Preparar transparències	10 días	mié 02/02/11	mar 15/02/11	172
174	Presentació - Assaig	8 días	mié 16/02/11	vie 25/02/11	173
175	Presentació - Defensa	0 días	lun 28/02/11	lun 28/02/11	165FC+60 días

FIGURA 14 – Llistat de subtasques que formen la tasca Presentació

Per a mostrar la distribució temporal de cadascuna de les activitats a realitzar, els següents diagrames de Gantt[4] la seva distribució dins el calendari, per a permetre un vista més detallada, igual com en el llistat de tasques anterior, primer es mostra la imatge de l'evolució temporal de les tasques principal i llavors un imatge referent al desenvolupament concret de cadascuna d'elles.

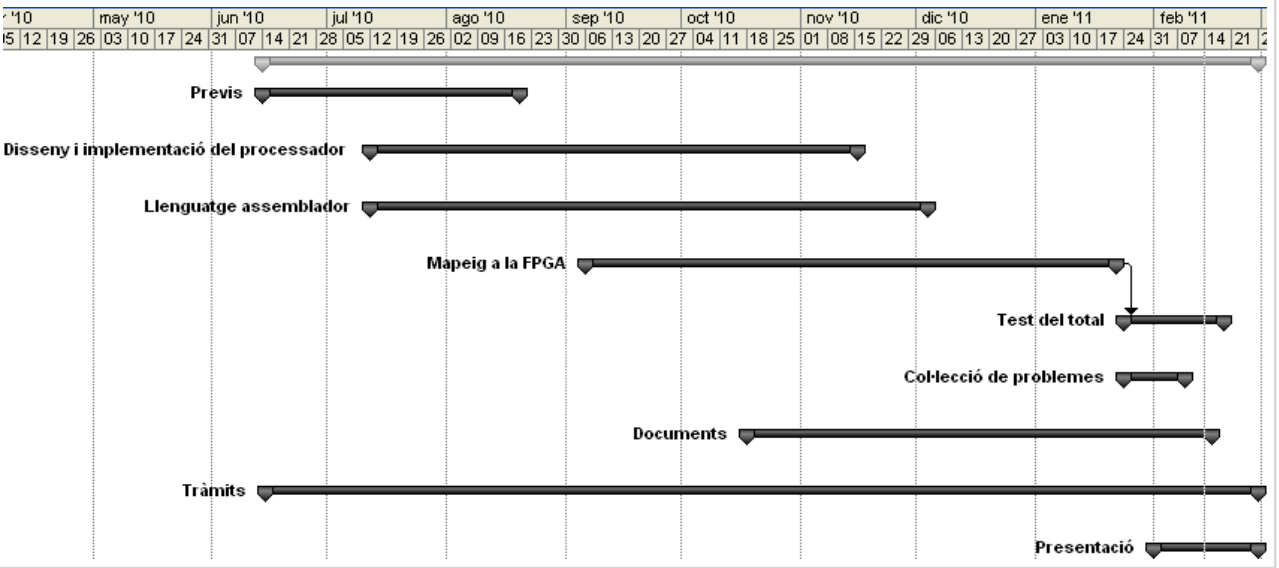


FIGURA 15 – Diagrama de Gantt que mostra la planificació i l'evolució temporal de les tasques principals del projecte.

[4] *Diagrama de Gantt: Eina gràfica per la planificació del desenvolupament de projectes, que mostra l'evolució temporal de les tasques que el componen.*

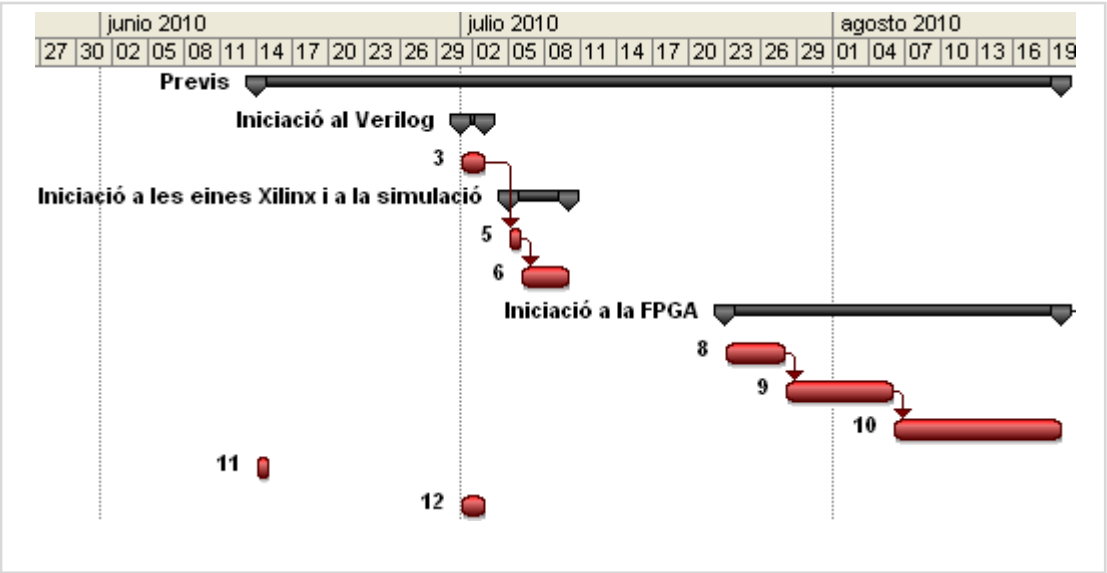


FIGURA 16 – Diagrama de Gantt que mostra la planificació i l'evolució temporal de les subtasques que formen part de la tasca Previs.

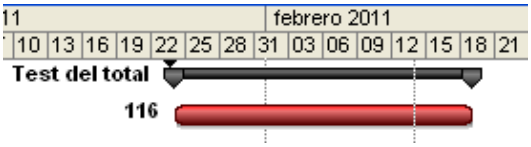


FIGURA 17 – Diagrama de Gantt que mostra la planificació i l'evolució temporal de les subtasques que formen part de la tasca Test Total.

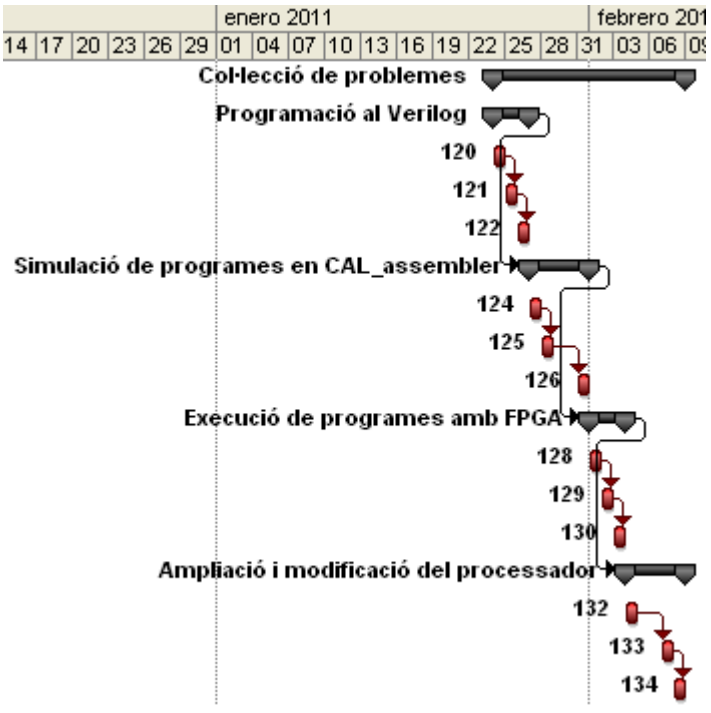


FIGURA 18 – Diagrama de Gantt que mostra la planificació i l'evolució temporal de les subtasques que formen part de la tasca Col·lecció de Problemes.

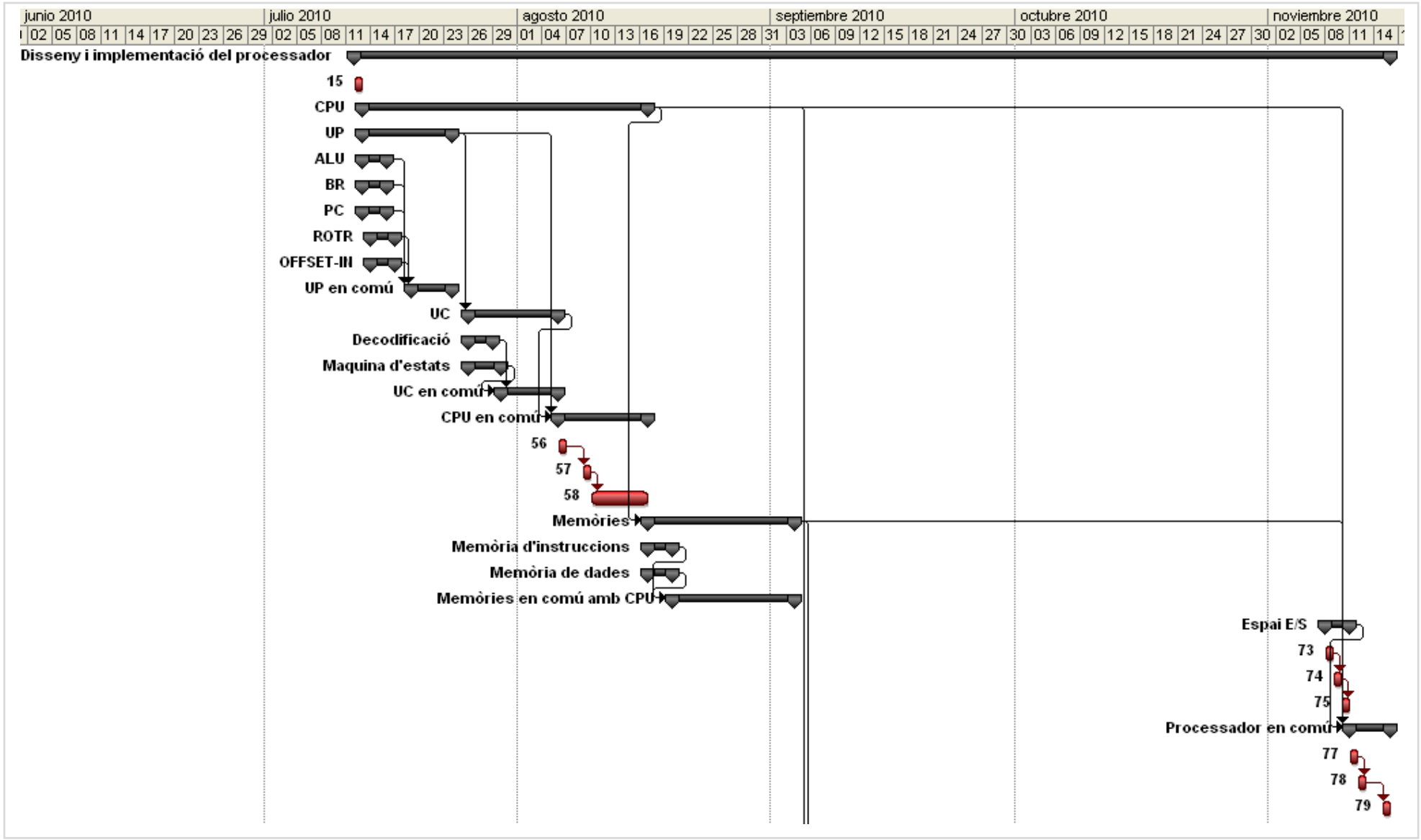


FIGURA 19 – Diagrama de Gantt que mostra la planificació i l'evolució temporal de les subtasques que formen part de la tasca Disseny i Implementació del Processador.

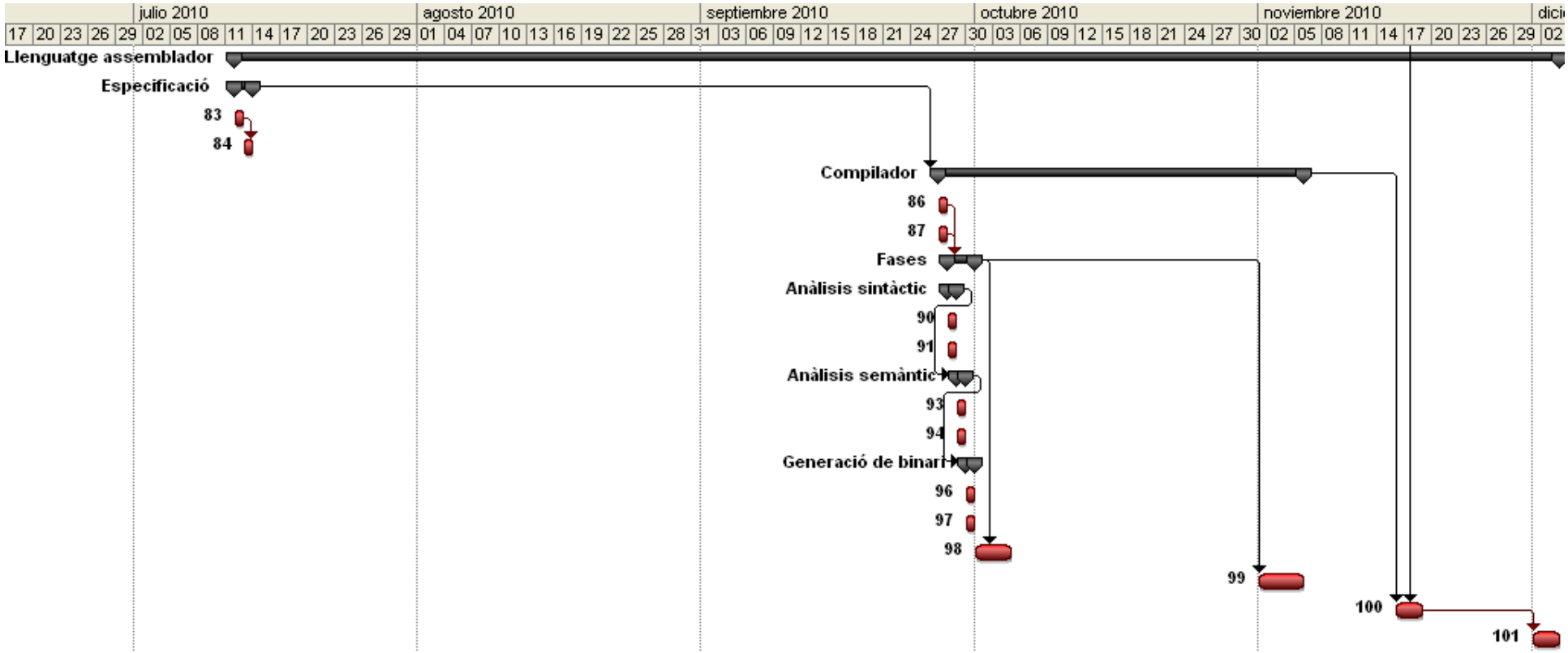


FIGURA 20 – Diagrama de Gantt que mostra la planificació i l'evolució temporal de les subtasques que formen part de la tasca Llenguatge Assemblador.

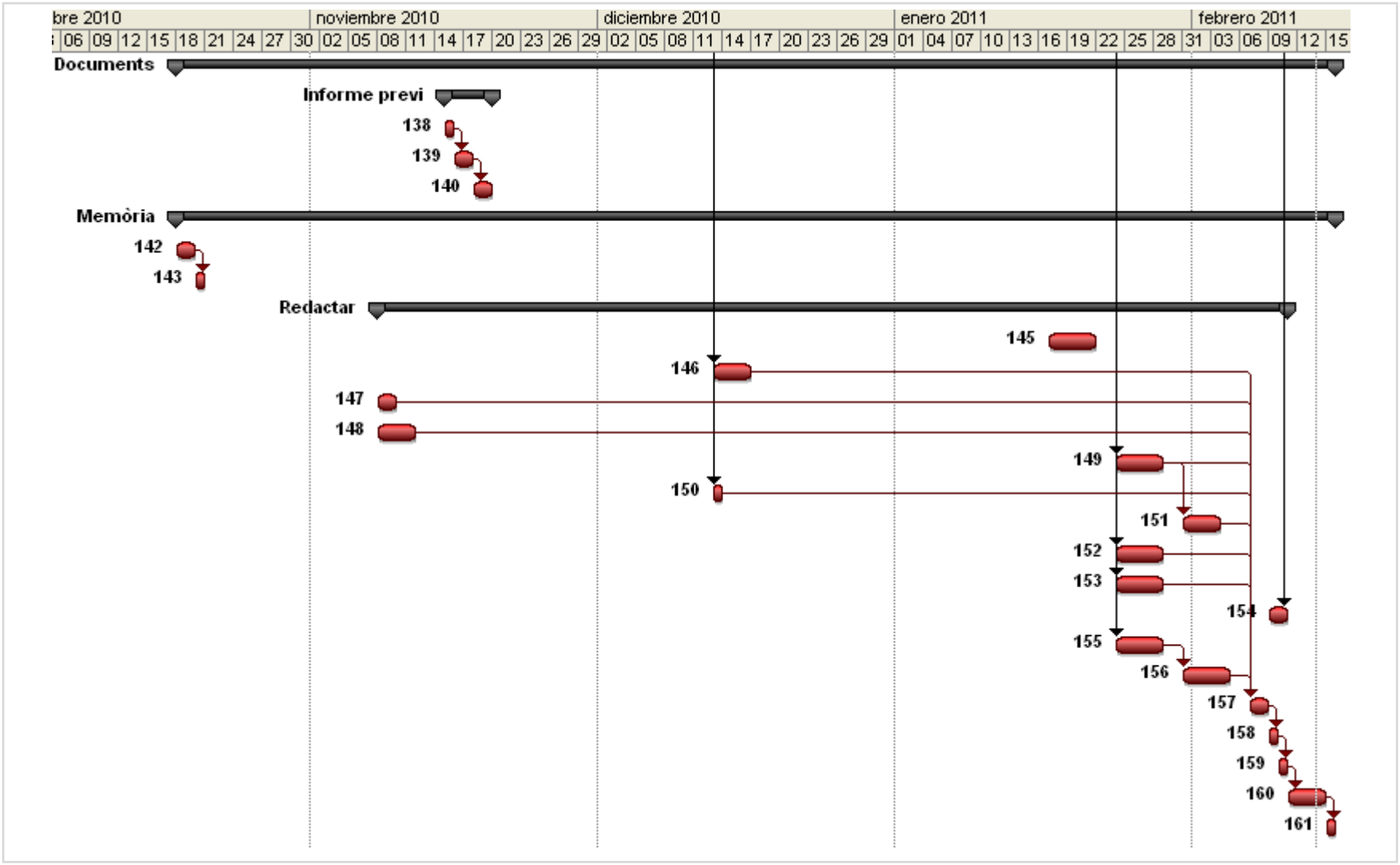


FIGURA 21 – Diagrama de Gantt que mostra la planificació i l'evolució temporal de les subtasques que formen part de la tasca Documents.

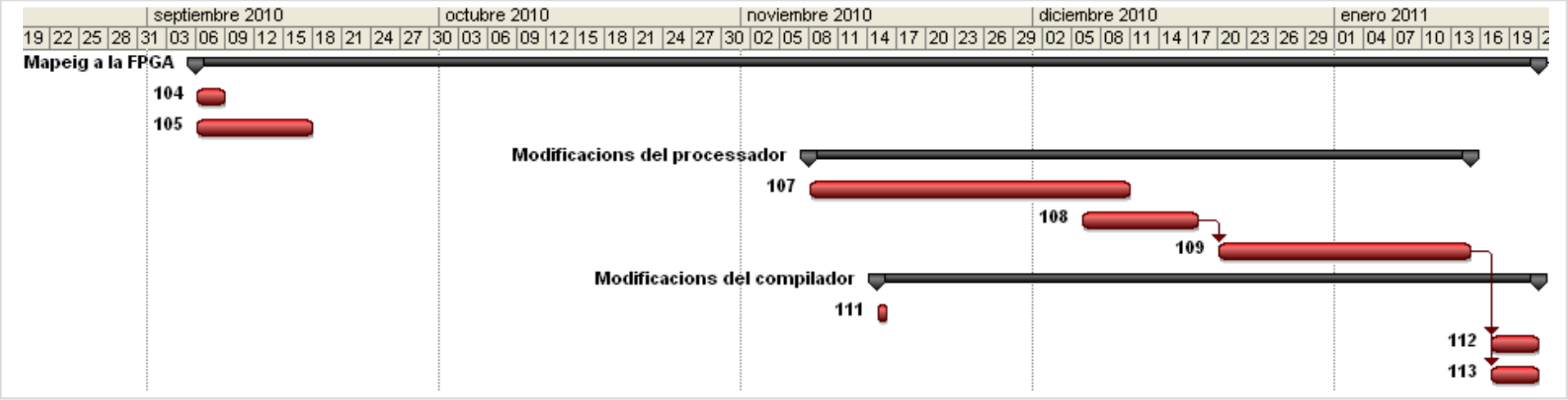


FIGURA 22 – Diagrama de Gantt que mostra la planificació i l'evolució temporal de les subtasques que formen part de la tasca Mapeig a la FPGA.

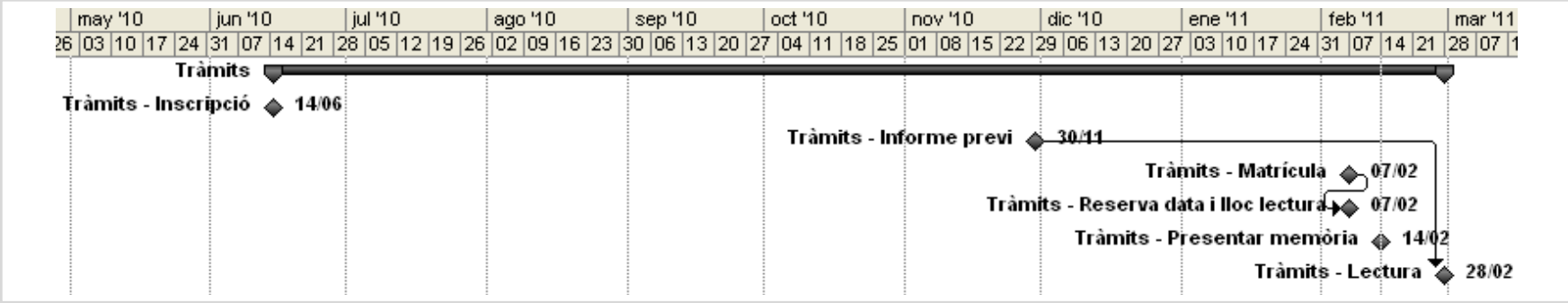


FIGURA 23 – Diagrama de Gantt que mostra la planificació i l'evolució temporal de les subtasques que formen part de la tasca Tràmits.

FIGURA 24 – Diagrama de Gantt que mostra la planificació i l'evolució temporal de les subtasques que formen part de la tasca Presentació.

ANÀLISIS DE COSTOS

Aquest apartat conté el detall del cost del projecte. Concretament analitza el cost de desenvolupament del projecte, que inclou les hores de treball i els recursos necessaris.

El projecte ha estat desenvolupat en la seva totalitat per una mateixa persona, així que no té sentit definir diferents rols als que assignar les tasques. El cost que suposa pel projecte l'únic recurs humà, és el determinat per a un becari de la FIB: **5,75 €/h.**

Un cop analitzada la planificació temporal del projecte, veiem que a grans trets, agafant un conjunt de tasques resum que engloben el conjunt de tasques definides en l'apartat anterior, es poden organitzar distribuïdes segons els mesos que ha durat el projecte.

Mes	Tasca Global	Dies
Juny	Definir Objectius	10
Juliol Agost	Formació i Estudi del tema	40
Setembre Octubre Novembre Desembre	Desenvolupament i Proves	90
Gener/Febrer	Documents i Memòria	35
Febrer	Acabats i presentació	10
	Durada Total Projecte	**185**

FIGURA 25 – Durada i distribució temporal de les tasques del projecte.

Vist això, de cara al cost del projecte, es suposa una jornada laboral de 6 hores al dia, on el preu l'hora del recurs és el definit al principi d'aquest apartat 5,75 €/h. Tant sols es comptabilitzaran aquelles hores dedicades al desenvolupament i proves i a la redacció dels Documents i de la Memòria del projecte:

Tasca	Dies	Hores	Preu
Desenvolupament i Proves	90	540	3105
Documents i Memòria	35	210	1207,5
	Cost Projecte		**4312,5**

FIGURA 26 – Cost de les tasques de desenvolupament del projecte.

Finalment tant sols queda afegir els costos pels recursos materials que s'han necessitat pel desenvolupament del projecte. Concretament ha estat necessari comprar una FPGA Spartan 3E, i aconseguir la llicència de les eines de Xilinx utilitzades durant tot el projecte. Pel que fa

aquesta última, és necessari realitzar una inversió inicial amb la compra d'un paquet que inclou 30 llicències.

Pel desenvolupament del projecte comptabilitzarem el cost de la compra total del paquet de 30 llicències, el que implica que en el moment d'implantar una nova assignatura, que és l'objectiu d'aquest projecte, llavors gràcies a aquesta inversió inicial, l'adquisició de llicències es considerarà gratuïta.

Recurs	Dolars	Euros
Llicències Xilinx (30 u)	999	739,26
preu unitat		33,3
FPGA Spartan 3E	149	110,26
Total Recursos		849,52

FIGURA 27 – Cost detallat dels recursos materials utilitzats en el projecte.

Per tant, finalment es pot deduir que el cost total de projecte, el qual inclou el cost del desenvolupament per part d'un becari, el cost dels recursos i la inversió inicial en llicències que en el moment d'implantació de l'assignatura serà un cost ja cobert, queda resumit a la següent taula.

Recurs	Cost
Cost Recursos Humans	4312,5
Cost Recursos Materials	849,52
Cost Total Projecte	5162,02

FIGURA 28 – Cost Total del Projecte.

RISCOS

En aquest apartat es llisten els riscos identificats en el projecte. Per a cadascun d'ells s'avalua el seu impacte, es detalla una estratègia de mitigació per intentar evitar-lo i es prepara un pla de contingència a seguir en cas de que es doni.

El projecte no està llest per la data d'entrega.

Impacte: *Baix.* El voler presentar-lo a principis de quadrimestre dóna un marge d'entrega de 4 mesos per davant.

Estratègia mitigació: Realitzar valoracions a nivell setmanal i mensual del treball realitzat per a valorar la dedicació necessària de cada setmana. Actualitzar el calendari del projecte en funció dels canvis.

Pla de contingència: En cas de no arribar a la data d'entrega, recalcular i replanificar les tasques pendents per a fixar una nova data d'entrega dins del quadrimestre.

Inexperiència en treball amb dispositius FPGA.

Impacte: *Alt*. Ja que és una part troncal del projecte i al desconèixer les seves funcionalitats inicialment pot complicar la seva utilització.

Estratègia mitigació: Planificar un temps d'aprenentatge conservador i realitzar valoracions a nivell setmanal i mensual del treball realitzat. Actualitzar el calendari del projecte en funció dels canvis.

Pla de contingència: En cas de no poder programar la FPGA amb el processador, prescindir d'aquesta part, i executar el processador tant sols a través del simulador de Xilinx que permet comprovar el seu correcte funcionament.

Incorporació de noves funcionalitats a mig projecte.

Impacte: *Mig.* Per cada nova funcionalitat demanada es pot decidir si aplicar-la o no, tot i que pot aportar un temps afegit d'estudi no contemplat inicialment.

Estratègia mitigació: Realitzar un bon disseny de les parts per que el fet d'aplicar canvis i aplicacions sigui senzill.

Pla de contingència: En cas de demanar-se una nova funcionalitat, avaluar-la i intentar portar a terme el canvi replanificant el projecte i en cas que sigui possible i si és necessari canviar la data d'entrega.

El termini d'entrega es veu reduït per una qüestió externa.

Impacte: *Alt.* El projecte s'ha d'entregar passi el que passi dins el quadrimestre de primavera de 2011.

Estratègia mitigació: Intentar en tot cas avançar tasques de les pròximes setmanes en cas que les planificades siguin més ràpides del esperat.

Pla de contingència: En cas d'aparèixer una causa externa que impliqui acabar el projecte abans del planificat, les tasques que prendran prioritat seran les definides als objectius i aquelles funcionalitats definides a mig projecte es finalitzaran en la mesura del possible.

CONCLUSIONS

CONCLUSIÓ TÈCNICA

El projecte "**Disseny i implementació d'un processador en un llenguatge de Descripció de Hardware"** es plantejava fa uns mesos com una proposta de múltiples objectius, però amb una finalitat molt concreta, la seva aplicació dins un entorn docent. Un cop finalitzat es pot afirmar que s'ha assolit aquest objectiu. Amb el projecte es disposa d'una base sòlida sobre la que treballar diferents objectes d'estudi relacionats amb el món del disseny dels processadors i la programació en llenguatges de descripció de hardware, concretament en Verilog.

Aquesta memòria posa a disposició un seguit de recursos, eines i documents, per a facilitar aquest enfocament dins l'àmbit de l'aprenentatge. A continuació, es fa un repàs de tots ells, esmentant en cada cas amb quin objectiu han estat escollits o desenvolupats.

EINES

- **Xilinx ISE Design Suite:** Eina que permet el desenvolupament de projectes en Verilog i comprovar la seva correctesa.
- **Xilinx ISE ISim:** Eina de que permet la simulació del comportament de projectes en Verilog.
- **Xilinx ISE iMPACT:** Eina de que permet la programació d'un dispositiu FPGA, amb projectes Verilog.
- **SPARTAN 3E:** Placa en la que es pot programar un projecte en Verilog.

DOCUMENTS

- **Manual d'Introducció a la Programació en Verilog**: Explica com utilitzar els seus components principals i es donen diferents exemples de programació. Això permet crear petits projectes en Verilog que es comportin d'una forma especificada.
- **Especificació i Disseny del Processador CAL16:** Conté l'especificació de les seves parts i del seu llenguatge, el *CAL_assembler*. Permet centralitzar o desembocar l'estudi sobre un processador senzill, estructurat i modular.
- **Manual d'Usuari:** Té com a objectiu facilitar l'aprenentatge de l'ús de les eines que presenta aquest projecte. Bàsicament s'hi defineixen els passos a seguir per a realitzar les principals accions necessàries alhora de desenvolupar projectes en Verilog.
- **Col·lecció de problemes:** Presenta un ampli ventall de problemes sobre els que focalitzar l'objecte d'estudi. A més cadascun d'ells ve acompanyat amb la seva solució.

Agafant com a base els recursos que proporciona aquest treball, de cara a organitzar una nova assignatura, cal definir els límits dins de cada àmbit, ja que les possibilitats d'implantació que presenta el projecte són múltiples.

- Se li pot donar un enfocament encarat al disseny de circuits i components electrònics, els quals es puguin implementar en Verilog i posteriorment provar mitjançant la FPGA Spartan 3E. Partint del desenvolupament dels diferents circuits, es podria anar evolucionant cap a la idea de processador i tancar amb la construcció del processador *CAL16* i el seu mapeig a la FPGA.
- Per altre banda, l'ensenyament es pot focalitzar, ja inicialment, sobre el processador *CAL16*, fet que permetria conèixer l'idea i l'estructura d'un processador, i alhora entendre i veure com estan implementats els seus components. A partir d'aquí, coneixent el processador en la seva totalitat, implementar modificacions i ampliacions del processador per a fer-lo més usable i complir amb l'objectiu principal del projecte, aprendre la programació en Verilog.

AMPLIACIONS

Aquest projecte queda obert a possibles modificacions i ampliacions. Concretament, algunes de les possibilitats d'ampliació i millora que, personalment, aportaria al projecte són:

- Ampliació de l'espai d'entrada sortida del processador, amb la possibilitat de connectar altres dispositius de la FPGA com podria ser la pantalla LCD[1] o el botó rotatori.
- Modificació del procés de càrrega de programes del processador, realitzant la càrrega de forma dinàmica a l'execució del processador a partir del bus sèrie de la FPGA.
- Ampliar el processador per aconseguir un conjunt d'instruccions més usable, que faci la programació en *CAL_assembler* més senzilla. I aprofitant la implementació d'instruccions més complexes evolucionar cap una màquina més elaborada i eficient, transformant-lo en un processador segmentat.

CONCLUSIÓ PERSONAL

Per acabar, com a conclusió més personal, m'agradaria fer un símil de la meva evolució durant el desenvolupament d'aquest projecte, i l'evolució que s'hauria de seguir, si en un futur aquest projecte és posa en pràctica dins un entorn docent.

Al inici d'aquest projecte, jo tenia una petita idea de com funcionaven els circuits electrònics i de quina havia de ser l'estructura d'un processador de les característiques del *CAL16*, però en canvi no tenia ni idea de com desenvolupar projectes en Verilog, ni de com programar una FPGA. Partint d'aquesta base, vaig començar programant en Verilog petits circuits que havia dissenyat, i veien el seu correcte funcionament a través de la simulació amb les eines de Xilinx. Un cop em vaig veure capaç de fer funcionar qualsevol circuit a nivell de simulació, vaig

[1] *LCD: Liquid Cristal Display. Pantalla plana de baix consum formada per píxels.*

endinsar-me dins el món de les FPGAs, dispositiu que em va aportar una visió molt més pràctica i real del que estava desenvolupant. En aquest moment vaig començar amb el disseny del processador *CAL16,* el qual un cop va estar implementat i funcionant correctament, em generava una sensació de superació, com més aplicacions li implementava, més millores a aplicar-hi em venien el cap, més ampliacions i més possibles exercicis interessants sobre els que treballar. Gràcies a totes aquestes idees, vaig desenvolupar la col·lecció de problemes i he definit possibles ampliacions del projecte.

El que vull transmetre amb aquesta descripció, és que aquest projecte no tant sols aporta un aprenentatge de forma evolutiva d'uns objectius definits, sinó que ben aplicat, pot aportar una motivació i un esperit de superació i millora dins el propi treball realitzat pels alumnes. I això és un aspecte molt important dins de l'àmbit docent, ja que aconseguir que els alumnes tinguin un interès i es sentin motivats, els dóna més ganes de treballar, més bona predisposició i com a conseqüència millors resultats acadèmics i millor assoliment dels coneixements.

BIBLIOGRAFIA

ANTLR [en línia]. Disponible a la Web: < http://es.wikipedia.org/wiki/Antlr >

CORTADELLA, Jordi. *Transparències curs CL de la Facultat d'Informàtica de Barcelona.*

CORTADELLA, Jordi. *Transparències curs PRO1 de la Facultat d'Informàtica de Barcelona.*

GODOY, Guillem i FERRER CANCHO, Ramon. *Parsing and AST construction with PCCTS.*

LEE, James M. *Verilog QuickStart - A Practical Guide to Simulation and Synthesis in Verilog - Third Edition.* Kluwer Academic Publishers. Dordrecht. ISBN: 0-7923-7672-2.

MAKEFILE IN WINDOWS [en línia]. Disponible a la Web: < http://edndoc.esri.com/arcobjects/ 9.0/ArcGISDevHelp/DevelopmentEnvs/Cpp/Makefiles/ Makefile.Windows.htm >

NAVARRO, Juan J. i JUAN, Toni. *Documentació curs de IC de la Facultat d'Informàtica de Barcelona.*

Project 4 – CAL16 implemented in LogiSim. Berkeley University. California.

SCRUM [en línia]. Disponible a la Web: < http://www.scrum.es/>

SCRUM [en línia]. Disponible a la Web: < http://es.wikipedia.org/wiki/Scrum >

SUTHERLAND, Stuart i MILLS, Don. *Verilog and System Verilog Gotchas – 101 Common Coding Errors and How to Avoid Them.* Springer. New York. ISBN: 0-387-71714-2.

XILINX. *Simulating a Xilinx 3.1 i CORE Generator Verilog Design.*

XILINX. *Memory Ingterface Solutions – User Guide.*

XILINX. *Using Block RAM in Spartan-3 Generation FPGAs*

XILINX. *Spartan-3E FPGA Starte Kit Board User Guide.*

XILINX. *Spartan-3 Generation FPGA User Guide – Extended Spartan-3A, Spartan-3E, and Spartan-3 FPGA Families.*

Disseny i Implementació d'un processador en un llenguatge de descripció de hardware

Memòria Tècnica

INTRODUCCIÓ

Aquest capítol anomenat Memòria Tècnica té com a objectiu principal descriure el disseny del processador *CAL16,* la integració de les seves parts i mostrar-ne el seu funcionament. Per aconseguir-ho s'ha organitzat de la següent manera.

Primer, en l'apartat anomenat Processador *CAL16*, es troba l'especificació del processador, la seva descripció, es fa un recorregut pels seus components a nivell de disseny amb una breu explicació del seu funcionament, i es llista i justifica l'entorn de treball utilitzat per a la seva implementació.

A continuació, en el segon apartat, trobem l'especificació del llenguatge *CAL_assembler*, llenguatge assemblador que executa el *CAL16.* En aquest apartat s'especifica el llenguatge i el tipus d'instruccions que el formen. La informació més concreta sobre el ventall d'instruccions que ofereix el llenguatge es troba en l'apartat de Annexes, concretament en l'Annex 4.

El següent apartat és el del compilador. Aquest compilador ha estat implementat expressament per a reconèixer el conjunt d'instruccions que formen un programa escrit en *CAL_assembler,* el qual detecta els errors de programació i en cas de no existir, genera un fitxer binari que conté la representació binària del programa compilat. Aquest fitxer binari es col·loca a la carpeta de inicialització del *CAL16* on queda apunt per a executar.

Finalment es mostra un conjunt de programes en *CAL_assembler* executats en el *CAL16* amb l'objectiu de demostrar el correcte funcionament de totes les seves parts. En cada exemple es parteix de la programació en assemblador del programa, es mostra el fitxer binari resultant un cop compilat i finalment els resultats de la seva execució. Aquest resultats es poden veure a través de dues vies:

- **Simulació:** On s'executa el programa utilitzant una eina de simulació, i en la que es poden consultar els valor temporals de tots el busos i senyals del *CAL16* en qualsevol moment de l'execució, gràcies a la generació d'un cronograma temporal que ho mostra des de l'instant inicial.
- **Execució en FPGA:** On el processador és sintetitzat i implementat mitjançant els circuits interns d'una *FPGA Spartan 3E*. En aquest cas els resultats del programes tant sols es poden veure reflectits mitjançant un component de la placa que inclou 8 leds, els que estan connectats com a dispositius de sortida del processador. A més els programes també poden dependre d'una entrada de dades directament connectada als 4 interruptors dels que disposa la placa.

A més, el tercer annex, conté un CD adjunt[1], on es poden trobar els codis font del processador programats en Verilog, els del compilador i d'altres recursos que es citen durant aquest document.

[1] *CD Adjunt: Per a obtenir recursos adjunts a aquesta Memòria veure capítol Annexes, Annex 3.*

PROCESSADOR CAL16

DESCRIPCIÓ

El *CAL16* és un computador de 16 bits, de tipus RISC[1], que respon a l'execució de programes descrits en el seu llenguatge assemblador, el *CAL_assembler*.

El llenguatge *CAL_assembler* està format per un conjunt reduït d'instruccions que ens permeten realitzar operacions lògiques (com la AND, OR o XOR), aritmètiques (com la suma), de desplaçament de bits, de càrrega d'immediats, d'accés a memòria (tant de lectura com d'escriptura), i de ruptura de seqüència (les que poden ser relatives al estat del computador o absolutes i condicionals o incondicionals).

El computador s'estructura en un espai de Memòria i una Unitat Central de Procés. En l'espai de Memòria s'hi troba la Memòria de Dades, els Registres d'Entrada Sortida i la Memòria d'Instruccions. La Unitat Central de Procés conté el centre de control i d'operacions del processador.

El *CAL16* funciona a una velocitat determinada pel senyal de rellotge, clk, i disposa d'un senyal de reinici, anomenat reset, amb la que es reinicia l'execució del programa carregat. Com a entrada i sortida de dades disposa de dos ports mapejats en l'espai de Memòria de Dades, un directament connectat a la sortida a través dels leds i l'altre a l'entrada dels interruptors.

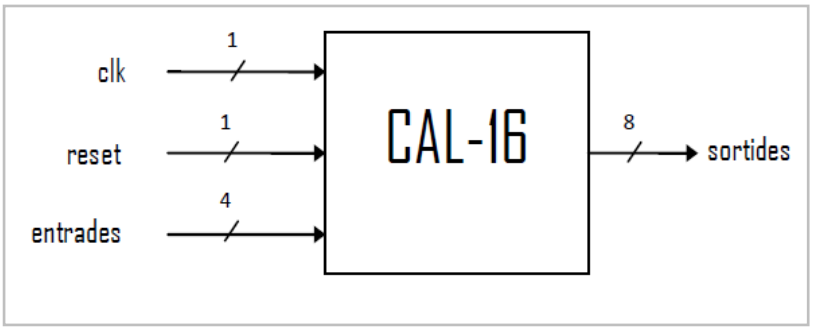


FIGURA 29 – Boc Processador CAL16

[1] *RISC: Reduced Instruction Set Computer. Tipus de microprocessador format per un conjunt reduït d'instruccions.*

DISSENY

Tal i com s'ha presentat en l'apartat anterior, es pot considerar el *CAL16* com la síntesi d'un conjunt de blocs on s'implementa cadascuna de les seves funcions principals. Bàsicament els blocs que el formen són:

- La Unitat Central de Procés (CPU) que és la part encarregada de l'execució. En aquesta es controla des de l'estat del processador i la màquina d'estats, que el guia en la execució de programes, fins al emmagatzematge dels valors temporals dels programes a executar i el conjunt de blocs que dóna els resultats de les diferents operacions. Per estructurar aquestes tasques la CPU està formada per la Unitat de Procés i la Unitat de Control.
- L'Espai de Memòria està format per una Memòria de Dades, 2 registres on a connectar un dispositiu d'entrada i un de sortida i una Memòria d'Instruccions. La Memòria de Dades i l'Espai d'Entrada/Sortida comparteixen busos, ja que les instruccions d'accés són les mateixes. Per a diferenciar una lectura del port d'Entrada de dades o d'una posició concreta de la Memòria de Dades, ho determinarem mitjançant l'adreça d'accés; les adreces a partir de la 0x0100, està directament connectat als diferents registres associats a dispositius d'entrada/sortida descrits en el projecte.

El processador inclou la connexió dels diferents busos d'entrada sortida a cadascun d'aquest blocs per aconseguir un correcte funcionament. En els apartats posteriors es pot trobar una descripció detallada de cadascun d'ells.

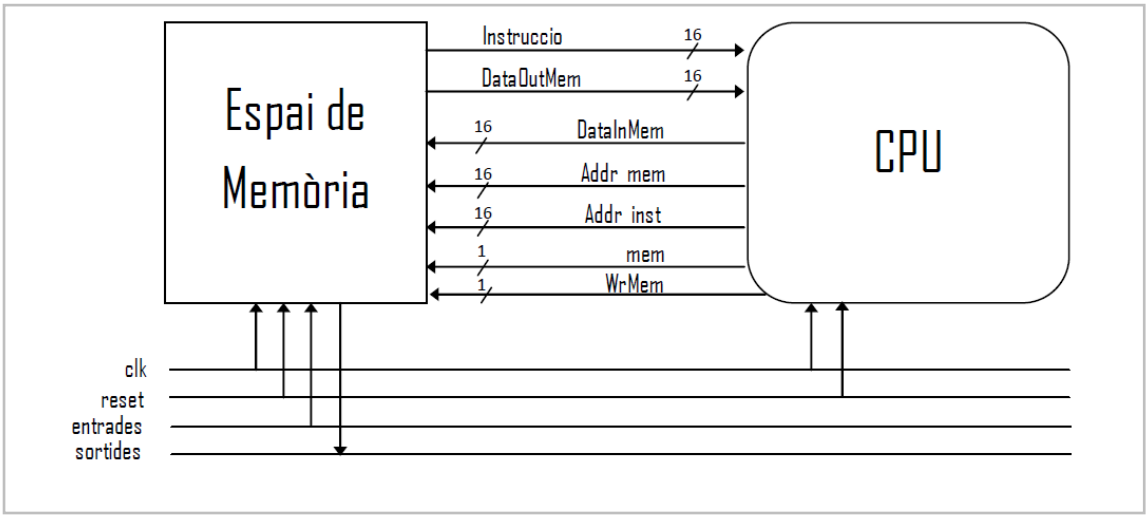


FIGURA 30 – Implementació interna a nivell de blocs del Processador CAL16.

Unitat Central de Procés

La Unitat Central de Procés té 2 busos d'entrada i 3 de sortida. Pel que fa als busos de sortida, és l'encarregada de generar l'adreça de la Memòria d'Instruccions de la nova instrucció a executar, i aquesta es dóna a través del bus de sortida AddrInst de 12 bits, també es dóna a través dels busos Mem_data i AddrMem, l'adreça de la Memòria de Dades en la que es vol llegir o escriure i en aquest últim cas el valor amb el que s'actualitzarà.

També té com a sortides d'un sol bit, el senyal mem que identifica l'estat d'accés a memòria del processador, i el senyal wrMem que indica que és una instrucció d'escriptura a Memòria de Dades. Aquests estan generats per la Unitat de Control.

Com a entrada, rep el valor de la nova instrucció a executar codificada segons el llenguatge *CAL_assembler* mitjançant una tira de 16 bits, a través del bus Instrucció de 16 bits. I en el bus Data_mem de 16 bits, el valor de la lectura corresponen a la posició Addr_mem de la Memòria de Dades en cas de tractar-se d'una instrucció d'aquest tipus.

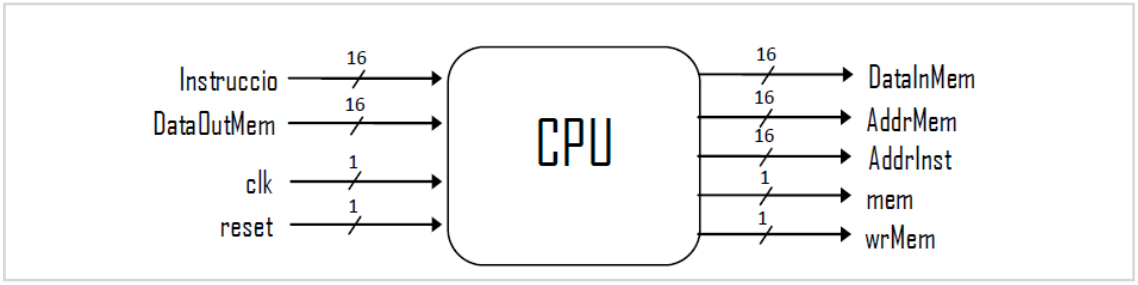


FIGURA 31 – Bloc Unitat de Procés Central

La Unitat de Control i la Unitat de Procés es comuniquen a través d'una Paraula de Control de 23 bits, i el bus Offset, en el que en funció de si la instrucció consta d'un operand immediat codificat en 4, 8 o 12 bits (depenent de la instrucció), hi trobem el valor d'aquest immediat codificat en 16 bits, sempre tractat com a enter codificat en Ca2[2], i amb el signe estès.

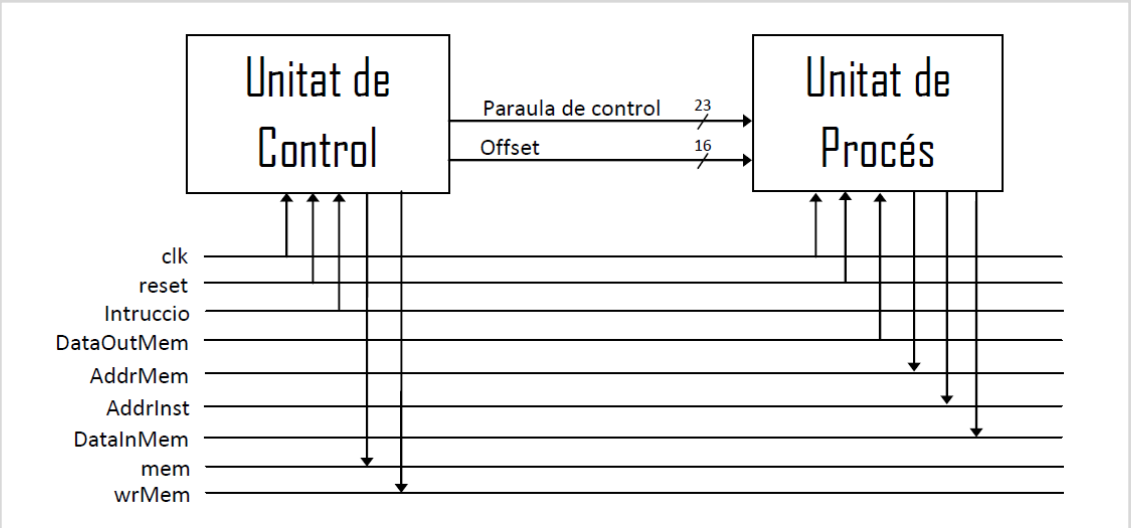


FIGURA 32 – Implementació interna a nivell de blocs de la Unitat de Procés Central

[2] *Ca2: Complement a 2, forma de representació de nombre enters en binari. Per més informació veure capítol Annexes, Annex 2.*

Unitat de Control [UC]

La Unitat de Control del *CAL16* és la màquina d'estats del computador juntament amb lògica interna que correspon a la generació de senyals de control que dirigeixen l'execució en la Unitat de Procés i les memòries en funció de la instrucció actual.

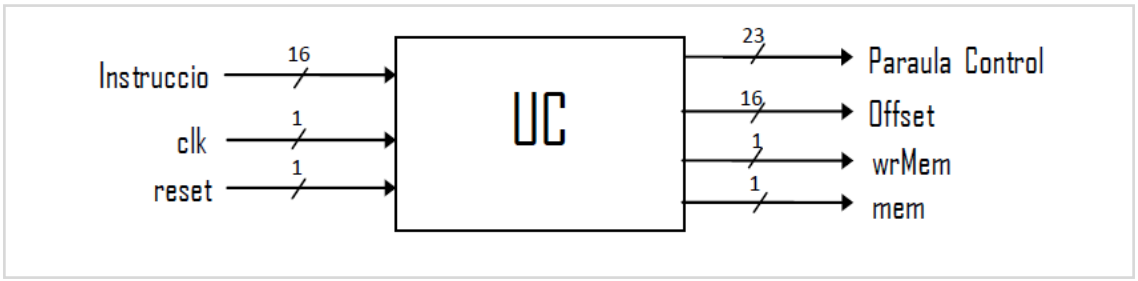


FIGURA 33 – Bloc Unitat de Control

La paraula de control és una tira de 23 bits formada pel conjunt de senyals que genera la Unitat de Control per encaminar les dades i controlar el funcionament del processador. A més d'aquesta paraula, com a bus entre els dos blocs també es genera l'immediat, de manera que el bus Offset pren el valor de l'operand immediat, ja sigui codificat en 4, 8 o 12 bits en funció de la instrucció, aquest es tracta com a enter codificat en Ca2 i se li estén el signe fins a completar els 16 bits. El bus mem, indica l'etapa d'accés a memòria del processador i s'utilitza com a bit d'activació de l'espai de memòria i el wrMem coincideix amb el bit de permís d'escriptura a la Memòria de Dades.

op	op	op	alu_rotr	b_off	off4_off8	off12	jmp	bz	bneg	wrMem	up_mem	RAenRD	RdenRB	RAenRB	wRd	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	AND
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	OR
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	XOR
0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	ADD
0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	ADDI
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	ROTR
0	1	1	0	1	0	0	0	0	0	0	1	0	0	0	1	LD
0	1	1	0	1	0	0	0	0	0	1	0	0	1	0	0	ST
1	0	0	0	1	1	0	0	0	0	0	0	1	0	0	1	LI
1	0	1	0	1	1	0	0	0	0	0	0	1	0	0	1	LIH
1	0	0	0	0	1	0	0	0	1	0	0	0	0	1	0	BNEG
1	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	BZ
1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	JR
1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	JMP
1	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	JMPI
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-

FIGURA 34 – Contingut Rom paraula de control

Com a entrades a la Unitat de Control, arriben els senyals de rellotge clk, que determina la velocitat del processador, i el senyal de reinici d'execució de programa, reset, que permet tornar a executar el programa carregat des de la primera instrucció. A més, pel bus de 16 bits anomenat Instruccio, arriba directament des de la Memòria d'Instruccions la següent instrucció a executar, codificada en una tira de 16 bits, que especifica el llenguatge *CAL_assembler*.

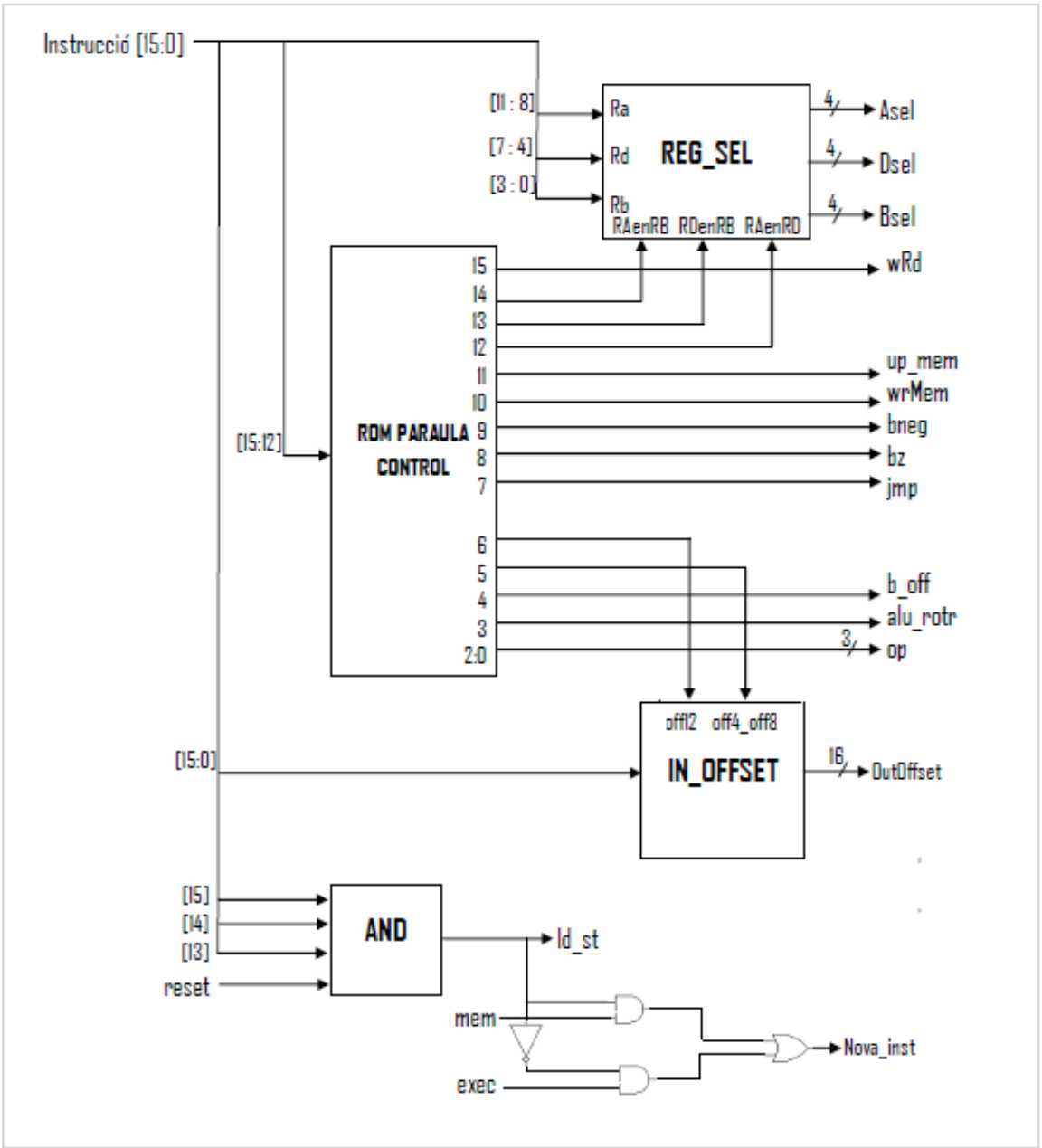


FIGURA 35 – Implementació interna a nivell de blocs de la Unitat de Control

Màquina d'Estats

El *CAL16* controla la seva execució a partir d'una màquina d'estats implementada dins la Unitat de Control, que estructura els seus passos en 4 etapes principals:

- ***reset*:** Etapa en la que tots els registres del processador s'inicialitzen amb el valor 0, per a començar l'execució des de la primera instrucció del programa (per definició la que es troba en l'adreça 0x000 de la Memòria de Instruccions). Estat d'espera d'inici d'execució.
- ***fetch*:** Etapa en que a partir de la nova direcció generada per Comptador de Programa PC, la Memòria d'Instruccions serveix la Instrucció a executar codificada en 16 bits.
- ***exec*:** Etapa en la que es descodifica la instrucció presenta la Unitat de Control, generant així la paraula de control que arriba a la Unitat de Procés, encarregada de guiar l'execució d'aquesta.
- ***mem*:** Etapa d'accés a Memòria de Dades, ja sigui per lectura o per escriptura, en cas de tractar-se d'una instrucció d'accés a memòria.

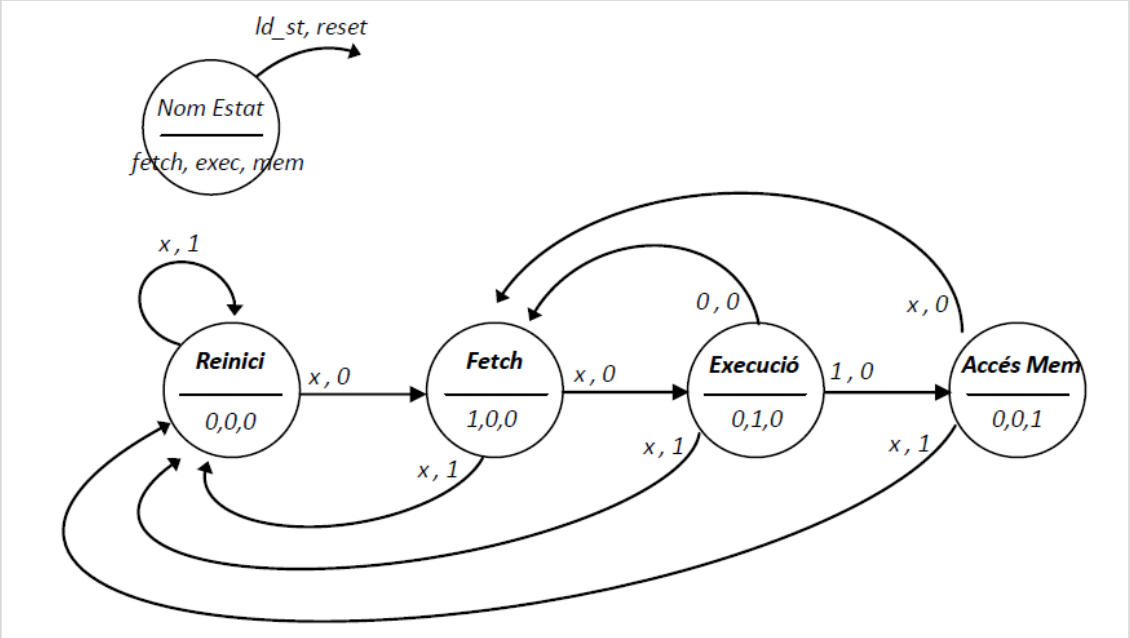


FIGURA 36 – Graf d'estats que implementa la Màquina d'estats que guia al Processador CAL16

Al inici de l'execució ens trobem a l'estat de *reset*, seguint la freqüència marcada pel rellotge, a cada flanc ascendent es realitza un canvi d'estat. Des de qualsevol estat, en cas que el senyal de *reset* valgui 1, es tornarà a aquest estat inicial reiniciant l'execució del programa.

Un cop el *reset* val 0, es realitza el fetch de la primer instrucció, que correspon en agafar la direcció generada pel comptador de programa (PC), el qual inicialment val 0, i accedir a la Memòria d'Instruccions per obtenir la instrucció emmagatzemada en aquesta posició. Llavors, indistintament i sempre i quan el *reset* no s'activi, es passa a l'estat de *exec,* que és el moment

en que la Unitat de Control descodifica la instrucció que li arriba i genera les senyals de control necessàries que permeten l'execució d'aquesta a la Unitat de Procés. Arribats en aquest punt podem seguir dos camins diferents, si la instrucció actual es tracta d'una instrucció d'accés a Memòria de Dades, aleshores el senyal *ld_st* s'activarà i el següent estat a executar serà l'etapa *mem* que permet realitzar aquest accés utilitzant la informació generada per la Unitat de Procés en l'etapa anterior, i per tant ja estable. En cas de no activar-se el senyal de *ld_st*, significarà que es tracta d'una operació simple, així que es procedirà a buscar la nova instrucció a executar tornant a l'etapa de fetch.

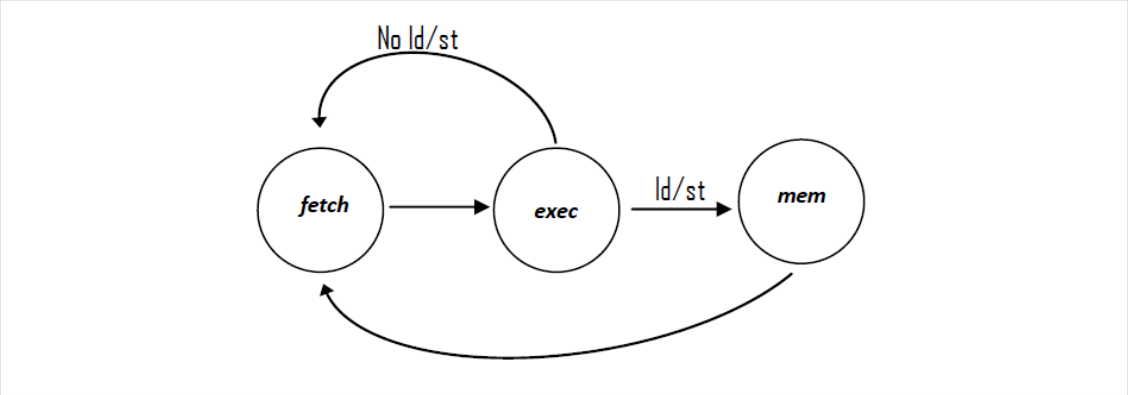


FIGURA 37 – Diagrama d'estats simplificat d'una instrucció del CALI6

Pel que fa a la implementació d'aquesta màquina d'estats amb components físics, cal definir els següents conceptes:

- **Estat actual**: Estat en el que ens trobem en el moment concret, pot ser reinici, fetch, exec, mem i es codifica amb 2 bits.
- **Estat següent**: Estat que es carregarà després del flanc ascendent del rellotge.
- **Taula de Transicions:** Rom de 16 paraules de 2 bits, que dona valor als bits de l'estat següent a executar, en funció de l'estat actual i de les senyals reset i ld_st (senyal que ens determina que la instrucció actual és una instrucció d'accés a Memòria de Dades i que per tant ha de disposar de l'etapa de *mem*).
- **Taula de Sortides:** Rom que en funció de l'estat actual determina l'etapa en la que es troba el processador. Dóna valor a les senyals de *fetch, exec* i *mem* que controlen l'execució del processador.

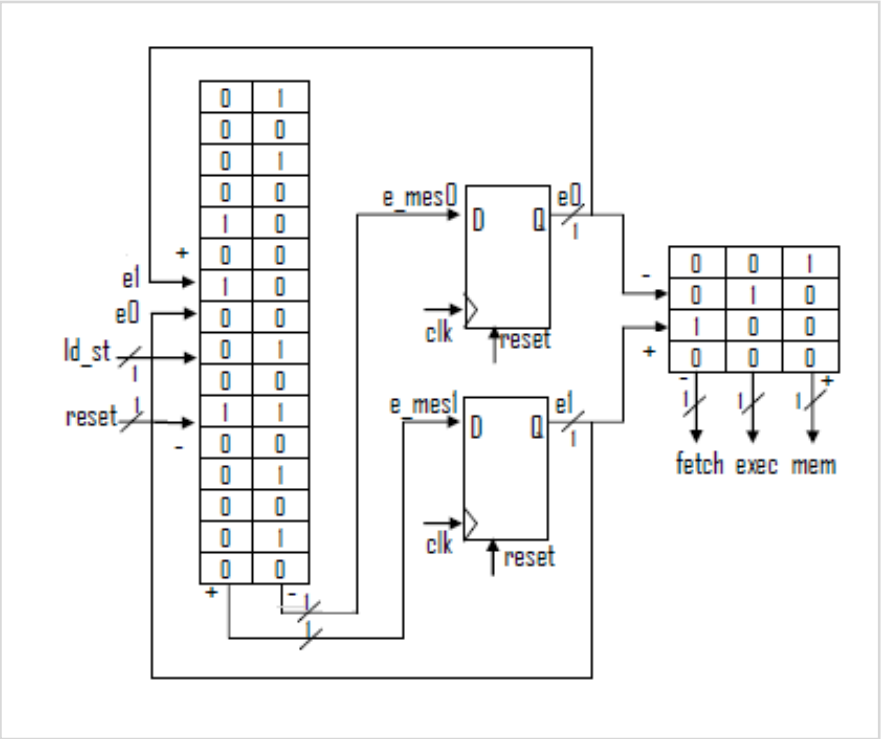


FIGURA 38 – Implementació interna de la Màquina d'estats del CAL16

Selector de Registres [REG_SEL]

És el conjunt de lògica combinacional que, en funció de la instrucció i els registres que conté codificats la tira de 16 bits que es troba en el bus d'entrada Instruccio, col·loca i encamina els seus operands, mitjançant els bits de control que determinen el registre a utilitzar en el bus correcte. Per identificar els registres dins la tira de bits que codifiquen una instrucció simple, es troba seguint l'ordre natural; Ra en els bits de pes 11:8, els Rb en els de 7:4 i els de Rd en els de 3:0.

Les sortides d'aquests és donen a través dels busos de 4 bits Asel, Bsel i Dsel, que formaran part de la paraula de control enviada a la Unitat de Procés per identificar els operand de la instrucció.

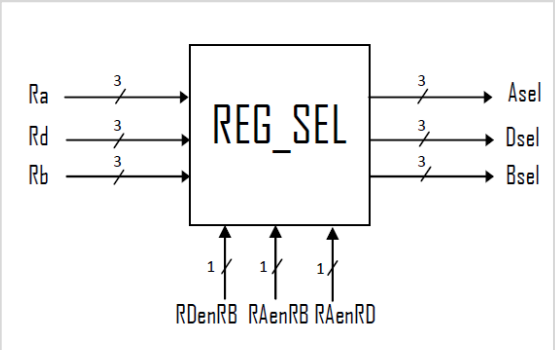


FIGURA 39 –Bloc Selector de Registres

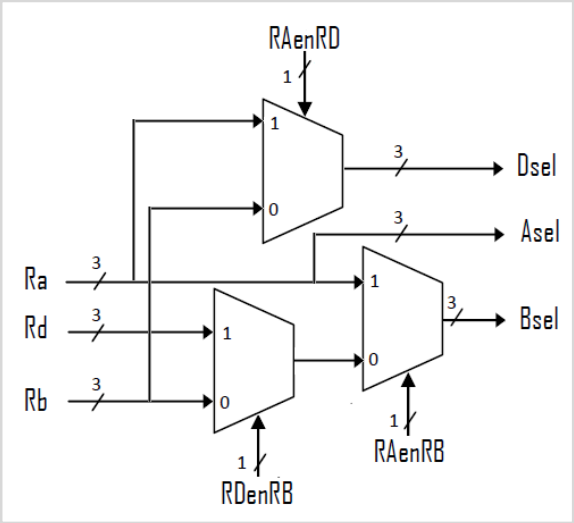


FIGURA 40 –Implementació interna del bloc Selector de Registres

Entrada d'Immediats [OFF_IN]

En funció de la instrucció disposem d'immediats de diferents rangs, des d'instruccions que tenen 3 registres com a operands, és a dir sense immediat, fins a d'altres que tenen un offset de 4, 8 o 12 bits.

Aquest bloc és l'encarregat d'encaminar el nombre de bits corresponents a l'immediat determinat segons la instrucció, cap al bus de sortida de 16 bits de la Unitat de Control, anomenat Immediat. Per a determinar el valor dels 16 bits de sortida, els bits seleccionats seran tractats com enters codificats en Ca2 i el seu signe serà estès fins a completar-los.

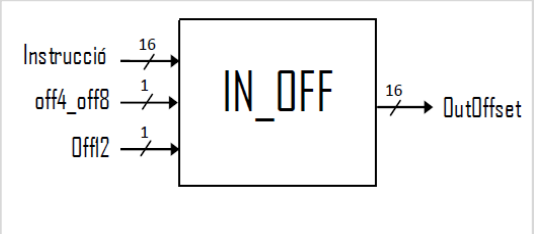


FIGURA 41 –Bloc Entrada d'Immediats

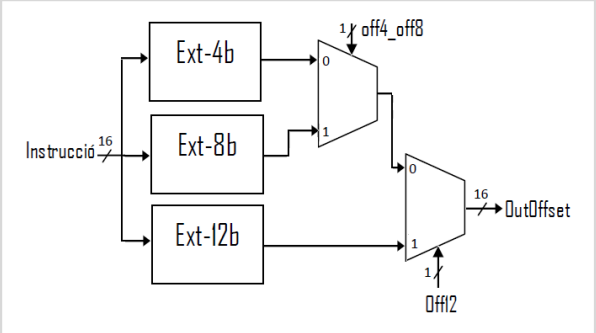


FIGURA 42 –Implementació interna del bloc Entrada d'Immediats

Unitat de Procés [UP]

La Unitat de Procés del *CAL16* es considera el nucli del processador, és l'encarregada d'emmagatzemar les dades temporals de l'execució dels programes, conté el conjunt de blocs necessaris per a realitzar les diferents operacions que identifiquen les instruccions del llenguatge *CAL_assembler*, i disposa de la lògica encarregada de generar i controlar el comptador de programa (PC) que indica la instrucció que s'està executant.

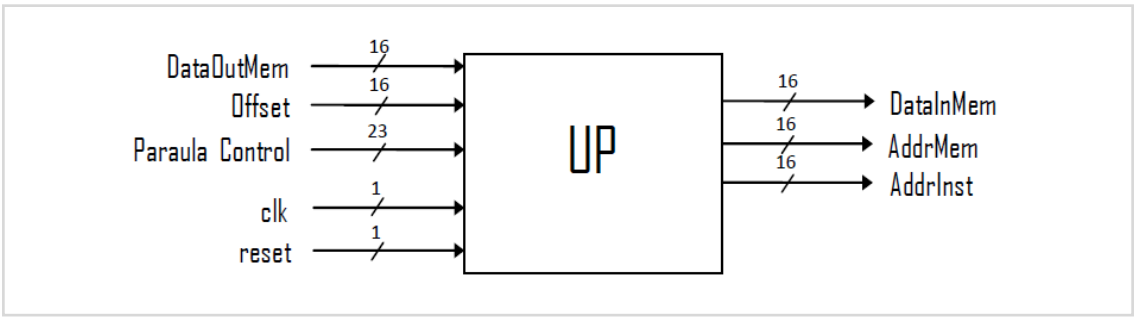


FIGURA 43 –Bloc Unitat de Procés

Per a un correcte funcionament de la Unitat de Procés, disposem de diferents multiplexors que encaminen les dades en funció de la instrucció que s'està executant, aquests multiplexors estan controlats a través de les senyals de control generades per la Unitat de Control descrites a continuació :

- **RA** [4 bits]: Bits que codifiquen el registre que es llegirà pel port A del banc de registres per a realitzar la operació que implica la instrucció que s'està executant.
- **RB** [4 bits]: Bits que codifiquen el registre que es llegirà pel port B del banc de registres per a realitzar la operació que implica la instrucció que s'està executant.
- **RD** [4 bits]: Bits que codifiquen el registre on s'escriurà pel port D del banc de registres la dada a emmagatzemar per la operació que implica la instrucció que s'està executant, en cas que aquesta actualitzi el valor d'algun registre.
- **op** [3 bits]: Valor codificat en binari que indica al bloc de la Unitat Aritmètico-lògica quina operació ha de realitzar.
- **alu_rotr** [1 bit]: Senyal encarregada de decidir si la dada vàlida per a la instrucció corrent és la que arriba des de la sortida del bloc ALU (alu_rotr=0) o des de la sortida del bloc ROTR (alu_rotr=1).
- **b_off** [1 bit]: Senyal encarregada de decidir si l'operand que utilitza la instrucció corrent és el que es troba en el bus B de sortida del Banc de registres (b_off =0) o és l'Offset d'entrada a la Unitat de Control (b_off=1).
- **up_mem** [1 bit]: Senyal encarregada de decidir si la dada vàlida per a emmagatzemar al registre destí del Banc de Registres per a la instrucció corrent, és la que arriba des de la sortida dels blocs que formen la Unitat de Procés (up_mem=0) o des de la Memòria de Dades (up_mem=1).

- **jmp** [1 bit]: Senyal que indica que la instrucció corrent és una instrucció de ruptura de seqüència no condicional (jmp=1).
- **bz** [1 bit]: Senyal que indica que s'està executant una instrucció de ruptura de seqüència condicional de tipus BZ, el que implica que la condició de salt es complirà en cas que l'operand emmagatzemat en el registre RB sigui igual a 0.

  ((bz==1 && RB==0) ? PC=PC+Ofsset : PC=PC+1)
- **bneg** [1 bit]: Senyal que indica que s'està executant una instrucció de ruptura de seqüència condicional de tipus BNEG, el que implica que la condició de salt es complirà en cas que l'operand emmagatzemat en el registre RB sigui un nombre negatiu codificat en Ca2.

  ((bneg==1 && RB<0) ? PC=PC+Ofsset : PC=PC+1)
- **jmp** [1 bit]: Senyal que indica que la instrucció corrent és una instrucció de càrrega d'immediat (li=1).
- **w_Rd** [1 bit]: Senyal que indica que la instrucció corrent emmagatzema el resultat del seu càlcul sobre el registre RD del Banc de Registres. És el senyal de permís d'escriptura sobre els registres (w_Rd=1).
- **nova_inst** [1 bit]: Senyal que indica que ens trobem a l'últim estat de la instrucció corrent. És a dir, si no és una instrucció d'accés a Memòria de Dades, s'activa durant l'etapa de *exec,* i en cas contrari s'activa durant l'etapa de *mem.* (nova_inst=1).

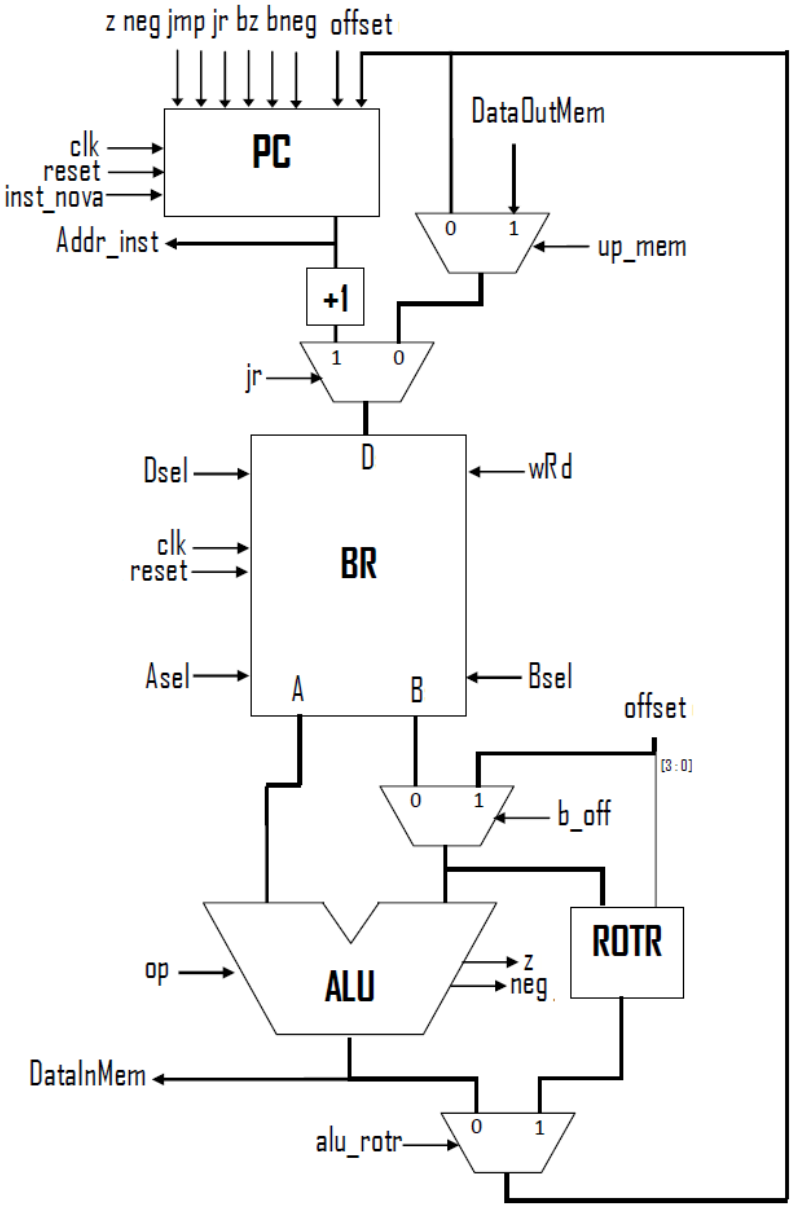


FIGURA 44 –Implementació interna a nivell de blocs de Unitat de Procés

Comptador de Programa [PC]

Bloc format per un registre anomenat PC encarregat d'emmagatzemar la direcció de la memòria d'instruccions corresponent a la instrucció que està executant. El PC també genera la direcció de la nova instrucció a executar durant la etapa de *fetch* del processador.

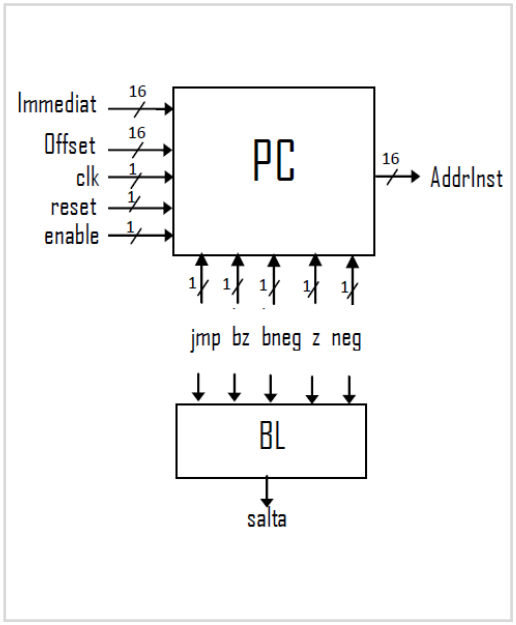


FIGURA 45 –Bloc Comptador de Programa i Bloc Lògica de Salt

Inicialment el PC s'inicialitza a 0, ja que es considera que la primera instrucció a executar és la que es troba a l'adreça 0 de la Memòria d'Instruccions. Després de cada instrucció el valor de PC s'incrementa en 1, per avançar a la següent instrucció a executar, sempre i quan la instrucció que s'està executant en aquest moment no sigui de salt (aquest increment s'anomena seqüenciament implícit). En cas d'execució d'una instrucció de salt, el bloc anomenat BL (Branch Logic o Lògica de salt) s'encarrega d'avaluar les condicions de salt i escollir el bus amb el nou valor que prendrà PC.

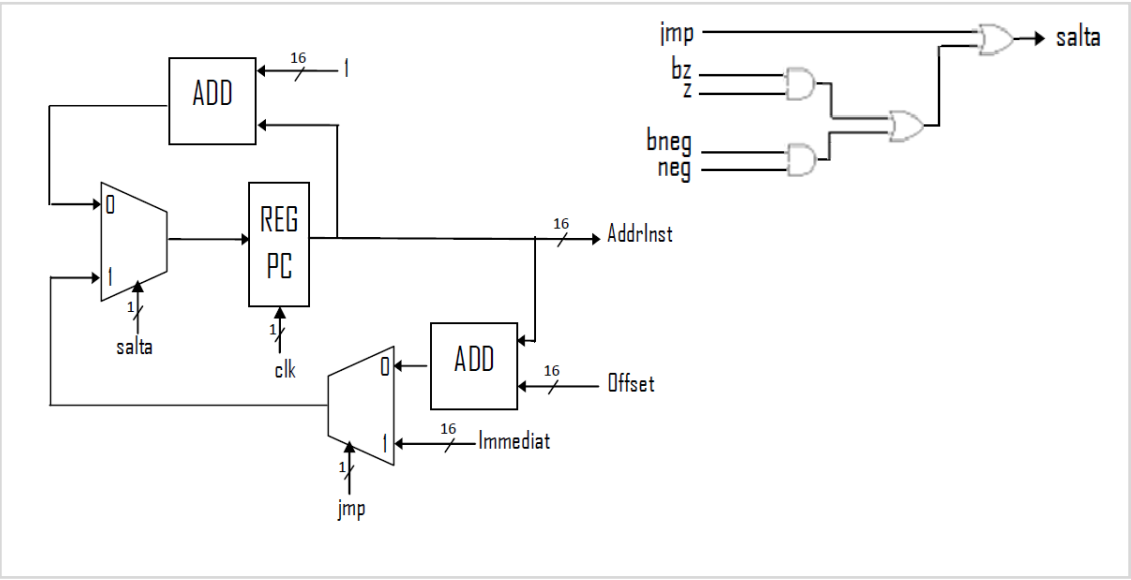


FIGURA 46 –Implementació interna del bloc Comptador de Programa i Lògica de Salt

Banc de Registres [BR]

És la part de memòria més propera i immediata a la CPU, el Banc de Registres del *CAL16* disposa de 16 registres de 16 bits cadascun, anomenats Rx on x={0..15}. Disposa de 2 busos de sortida (A i B) que permeten realitzar lectures de la informació emmagatzemada en el registre seleccionat segons la senyal de control Asel i Bsel respectivament. I d'un bus d'entrada (D) per a l'escriptura del valor D al registre indicat en Dsel.

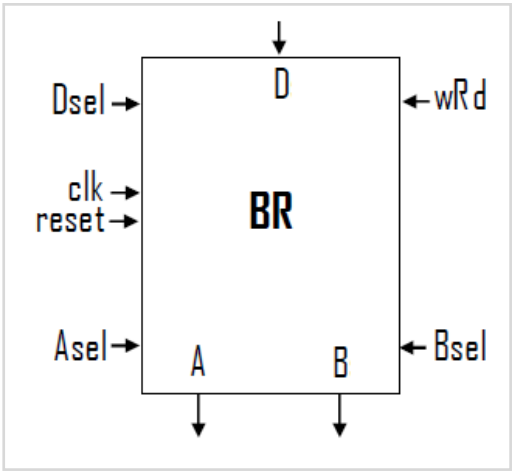


FIGURA 47 –Bloc Banc de Registres

La lectura i l'escriptura dels registres és síncrona amb la senyal de rellotge, tot i que l'escriptura també està controlada per una senyal de permís d'escriptura (wRd) que pren valor en funció de la instrucció que s'està executant, determinant amb un 1 si cal emmagatzemar el valor present en el bus D en el registre destí seleccionat per Dsel o si per contrari, amb un wRd=0 determinant que la instrucció executada no implica cap enregistrament del resultat a un registre. El moment d'escriptura és al final de l'execució de la instrucció, quan un cop a realitzat el càlcul, el resultat es troba estable en el bus D i s'emmagatzema al registre destí seleccionat. Aquest moment coincideix amb el inici de l'etapa de fetch de la següent instrucció.

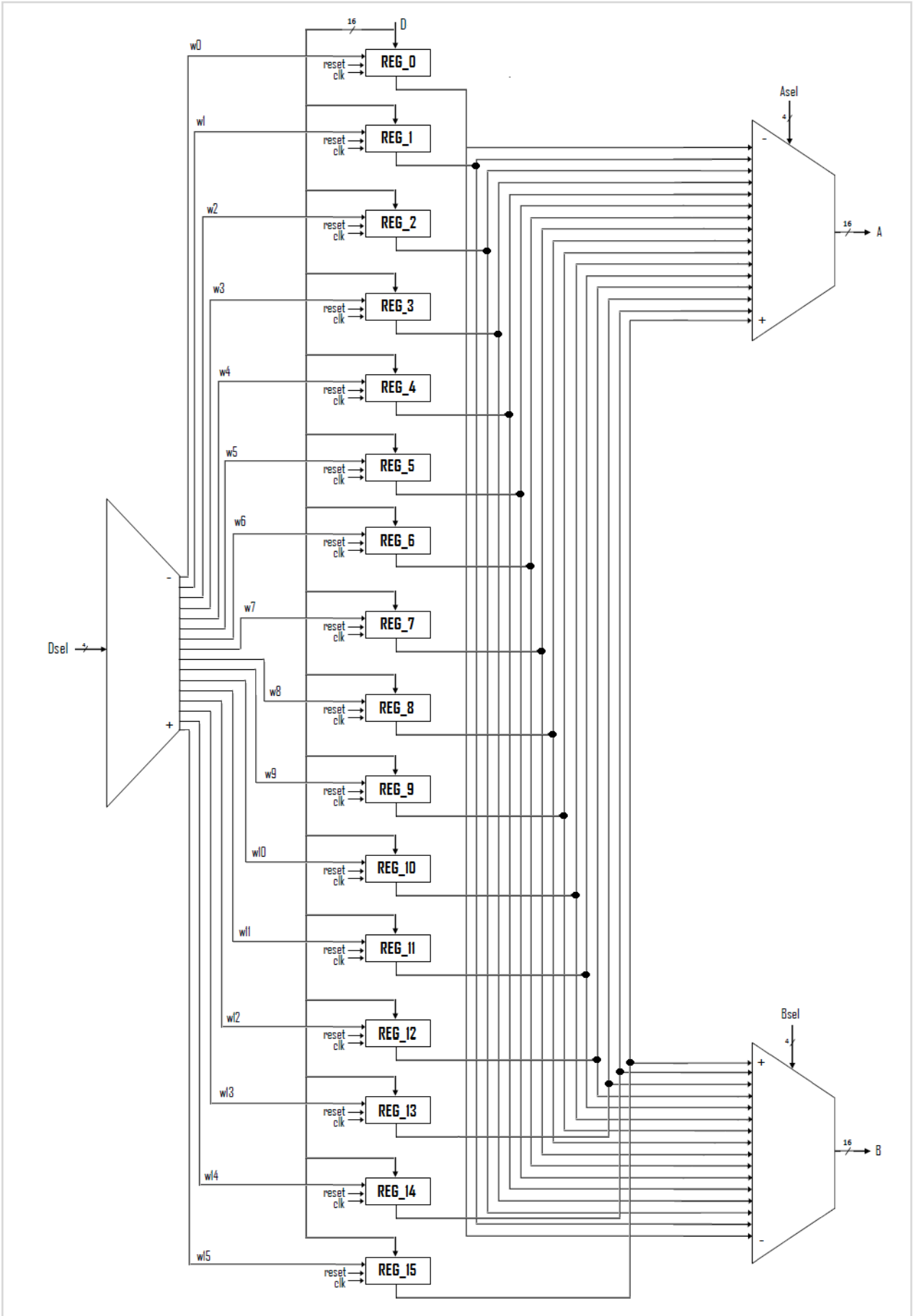


FIGURA 48 –Implementació interna del bloc Banc de Registres

Unitat Aritmètic-Lògia [ALU]

Circuit combinacional que encapsula la operació aritmètica de suma, les operacions lògiques de AND, OR i XOR, la càrrega d'immediats i dóna com a sortida informació referent al resultat obtingut amb l'operació seleccionada. Amb ella es permet l'execució i el càlcul del resultat de la instrucció que es troba en l'etapa d'execució.

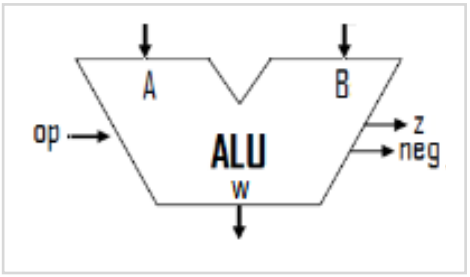
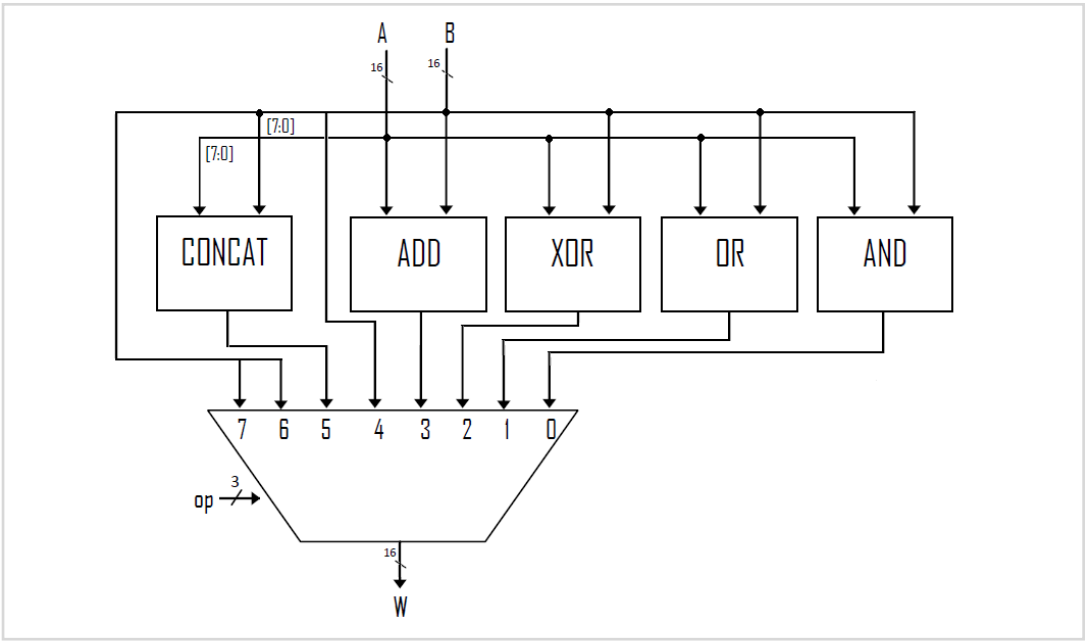


FIGURA 49 –Bloc Unitat Aritmètic.Lògica

El bloc de la ALU té dues entrades de 16 bits, per on arriben els operands de l'operació a realitzar, una senyal de control de 3 bits anomenada op que indica quina operació s'ha de realitzar, un bus de 16 bits de sortida per on es dóna el resultat de la operació, el qual s'estabilitza després del temps de propagació de la senyal op i els 2 busos d'entrada. També consta de dues senyals de sortida, que permeten al bloc PC (descrit anteriorment), controlar l'estat del processador en instruccions de salt condicionals.

Tot i disposar de 8 funcions a escollir generades pel senyal de control op de 3 bits, inicialment tant sols se'n utilitzen 6, les quals responen a les següents operacions:

- [op = 000] – Operació AND.
- [op = 001] – Operació OR.
- [op = 010] – Operació XOR.
- [op = 011] – Operació ADD.
- [op = 100] – Deixar passar el valor del bus d'entrada B.
- [op = 101] – Escriptura en la part alta del valor del bus d'entrada de A la informació representada amb els 8 bits de menys pes de l'entrada B.
- [op = 110] – No usades, destinades a modificacions de la màquina.
- [op = 111] – No usades, destinades a modificacions de la màquina.



Rotació [ROTR]

FIGURA 50 –Implementació interna a nivell de blocs de la Unitat Aritmètic-Lògica

El bloc combinacional ROTR disposa de 2 entrades (A i immediat), una de 16 bits i una altra de 4, respectivament. També té un bus de sortida de 16 bits (W) que pren el valor resultant d'executar un desplaçament d'un nombre determinat de bits, que l'indica l'immediat. Sobre els bits del bus A s'aplica el desplaçament de forma rotatòria, és a dir, el bit de més pes que al desplaçar-lo es surt de rang passa immediatament a ser el bit de menys pes aconseguint així un efecte rotatori sobre els bits del bus A.

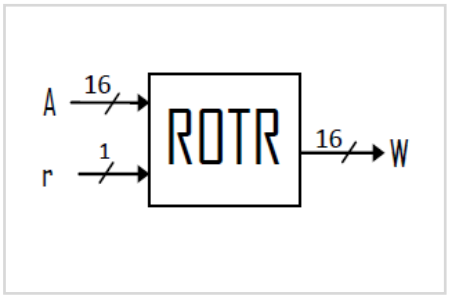


FIGURA 51 –Bloc Rotació

L'immediat que determina el nombre de bits a desplaçar es tractat com a un nombre natural codificat amb 4 bits, per tant permet fer desplaçaments dins del rang [0 .. 15], fet que permet el desplaçament a totes les possibles combinacions de nombres.

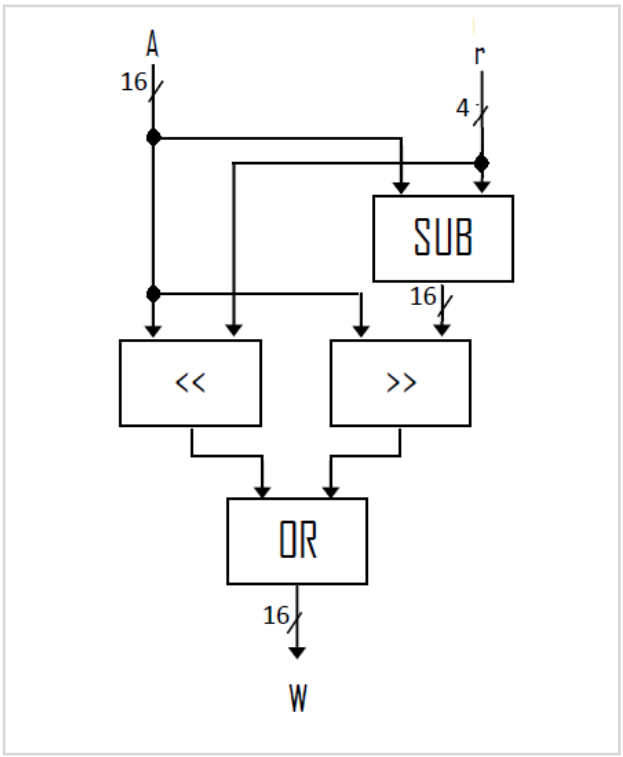


FIGURA 52 –Implementació interna a nivell de blocs del Rotació

Espai de Memòria

L'espai de Memòria del *CAL16* està dividit en tres components, la Memòria d'Instruccions (MI), la Memòria de Dades (MD) i l'espai d'Entrada-Sortida (REG). Per accedir a la Memòria d'Instruccions, s'utilitzen busos separats, ja que es tracta d'un processador de memòries no unificades, però en el cas de la Memòria de Dades i els Registres de E/S comparteixen els busos, d'aquesta manera en funció de l'adreça d'accés estarem accedint a les dades de Memòria o a les dels Registres, però podrem utilitzar les mateixes instruccions per a realitzar l'accés.

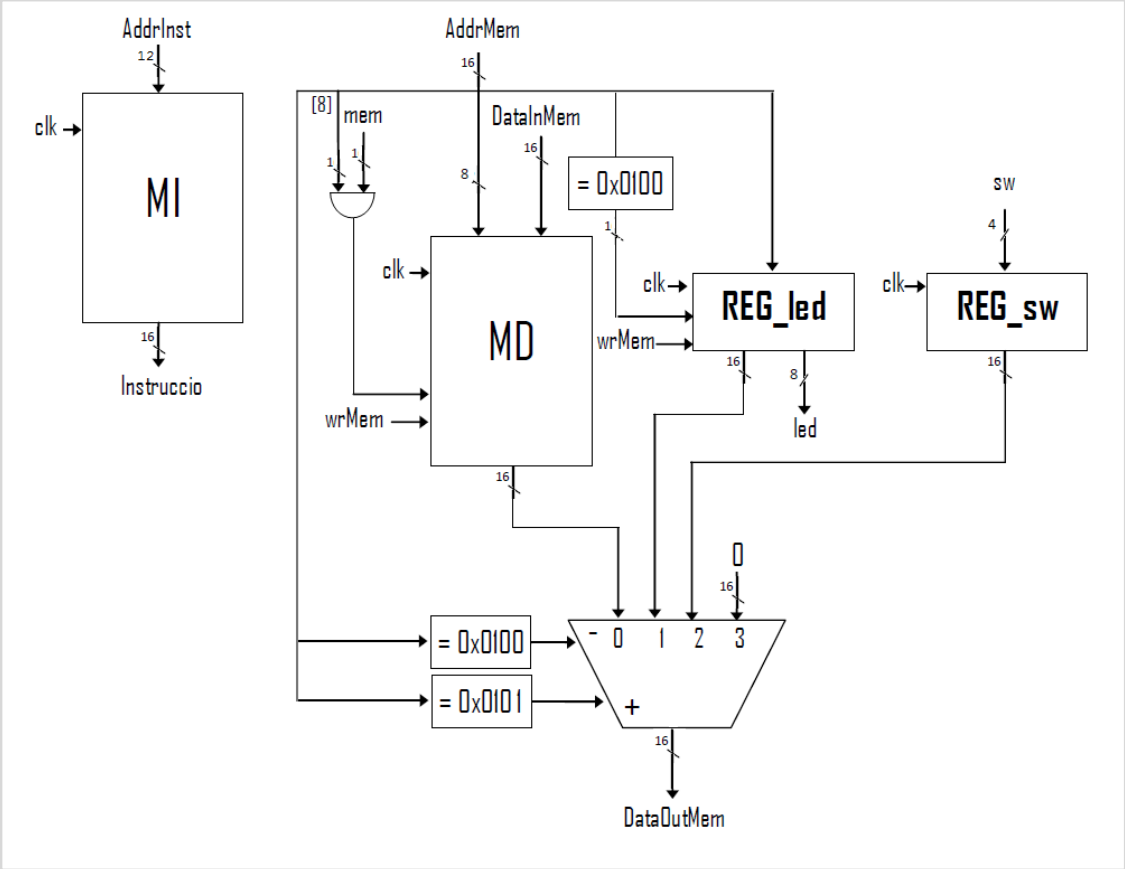


FIGURA 53 –Implementació interna a nivell de blocs de l'espai de Memòria del processador CAL16

Memòria d'Instruccions [MI]

La Memòria d'Instruccions, és una memòria ROM[3] formada per 4096 paraules de 16 bits cadascuna, on s'utilitza una adreça de 12 bits per a direccionar els accessos. És l'encarregada d'emmagatzemar el programa a executar codificat en binari en les seves cel·les, a cada direcció hi va una instrucció i la primera instrucció del programa es col·loca a la direcció 0x000, punt d'inici d'execució (al iniciar l'execució d'un programa, un cop es desactiva el senyal de reset, el Comptador de Programa PC val 0 indicant així que la primera instrucció a executar es troba a la direcció 0x000).

[3] *ROM: Read Only Memory, tipus de memòria electrònica de la qual només se'n pot llegir la informació.*

Al inici de l'execució d'un programa, la Memòria d'Instruccions s'inicialitza amb el valor binari de cada instrucció per files. Per a inicialitzar-se llegeix el fitxer anomenat *program.bin*, situat a la carpeta de programes del processador. Aquest fitxer conté les tires de bits que codifiquen cadascuna de les instruccions que formen el programa, a més de tires de 16 zeros per donar valor a les posicions no inicialitzades de la Memòria d'Instruccions, això és així a causa de la necessitat de definir tots els valors de la memòria alhora de mapejar-la en ja FPGA.

La Memòria d'Instruccions té un bus de 12 bits d'entrada, anomenat AddrInst, per on s'especifica l'adreça de la instrucció que es vol obtenir, i l'entrada de la senyal de rellotge ja que la lectura és síncrona. Com a sortida té el bus de 16 bits Instruccio, per on es serveix la instrucció associada a la direcció codificada en el bus AddrInst.

Memòria de Dades [MD]

La Memòria de Dades, és una memòria RAM[4] formada de 256 paraules de 16 bits cadascuna, s'utilitza una adreça de 8 bits per a direccionar els accessos. Permet accessos de lectura per a consultar el seu contingut donada la direcció d'una posició, o d'escriptura actualitzant el valor d'una posició de memòria en funció de l'adreça que la identifica i un nou valor donat.

La Memòria de Dades té un bus de 8 bits d'entrada, anomenat AddrMem, per on s'especifica l'adreça de la dada que es vol obtenir o actualitzar, en cas de tractar-se d'una escriptura, el nou valor a emmagatzemar en la direcció AddrMem el trobem en el bus d'entrada de 16 bits dataInMem. Disposa també de 3 entrades més de control, la senyal de rellotge ja que tant la lectura, com l'escriptura són síncrones, la de permís d'escriptura w, i la de selecció; ChipSelect, cs, que és l'encarregada d'activar la memòria sols en l'etapa mem dels possibles estats del processador, que determina que la instrucció realitza un accés a Memòria de Dades i que aquest no fa referència als Registres d'Entrada/Sortida .

Com a sortida té el bus de 16 bits dataOutMem, per on es serveix la dada demanada, la que és especificada en funció del valor de l'entrada AddrMem.

Espai d'Entrada - Sortida [E/S]

En l'Espai d'Entrada – Sortida, actualment hi ha definits 2 Registres, un d'escriptura associat a l'adreça 0x0100 i un de lectura a l'adreça 0x0101. Això és així, ja que actualment en el projecte només disposem de dos dispositius per a connectar en aquests ports. Si en un futur es realitzés una ampliació en aquest espai amb nous dispositius, això seria possible mitjançant les adreces a partir de la 0x0110.

Pel que fa al bloc de sortida d'aquest espai, consta d'un Registre de 8 bits. Aquest bloc disposa d'una sortida led de 8 bits amb el valor actual del registre la qual va directament connectada a la sortida leds del processador. D'aquesta forma, quan s'executi el processador mitjançant la FPGA, es pot veure en tot moment el valor d'aquest registre en els 8 leds dels que disposa la placa. Si el bit associat al led està actiu (és igual a 1) aleshores el led es mantindrà encès. A part d'aquesta sortida, disposa dels mateixos busos d'entrada i sortida que la Memòria de Dades, ja

[4] *RAM: Memòria d'Accés Aleatòri, tipus de memòria caracteritzada per permetre lectures i escriptures.*

que de la mateixa forma explicada anteriorment, podem consultar i modificar el valor del registre.

El bloc que permet l'entrada d'informació, està format per un registre de 4 bits associat a l'adreça 0x0101. Aquest pren el valor dels 4 bits d'entrada del processador anomenat *sw.* Aquesta entrada durant l'execució mitjançant la FPGA, anirà connectada als interruptors, així si un interruptor està actiu, en el seu bit associat del nostre Registre hi trobarem un 1. Aquest Registre només permet instruccions de lectura sobre ell i com en el cas anterior compartiex bus de lectura amb el Registre del leds i amb la Memòria de Dades. No permet escriptures ja que sempre disposa del valor que indiquin els interruptors, en cas d'intentar realitzar una escriptura sobre la seva direcció aquesta serà ignorada pel processador.

ENTORN, EINES I LLENGUATGE DE DESENVOLUPAMENT

Per a programar el *CAL16*, s'ha escollit un llenguatge de la família dels llenguatges de descripció de hardware (HDL) anomenat Verilog. Aquest ha permès implementar bloc a bloc cadascun del components que formen el Computador i aconseguir així un processador modular i canviable, amb l'objectiu de permetre incrementar o modificar les seves funcionalitats facilment. En algun moment es podria haver explotat més la potència del llenguatge per a realitzar operacions amb tires de bits, però s'ha decidit no aprofitar-se'n per a mantenir aquesta estructura interna de blocs i tenint en compte l'objectiu docent del projecte, permetre així l'estudi dels blocs combinacionals[5] i seqüencials[6] que el formen.

L'entorn utilitzat per a la implementació del *CAL16* ha estat: Xilinx ISE Design Suite. El qual disposa d'un conjunt d'eines, entre les quals, es permet la creació de projectes formats per blocs en Verilog, el reconeixement del llenguatge, la seva posterior correcció de sintaxis i simulació de comportament, o síntesi i implementació per poder executar-ho mitjançant una FPGA.

El Sistema Operatiu sobre el que s'ha treballat és Windows, ja que es tractava d'un requisit inicial del projecte. Per aquest motiu, aquest i tots els demés components que es presenten en el projecte s'executen sobre el mateix sistema operatiu, per a facilitar la prova i execució del processador.

[5] *Circuit Lògic Combinacional: Sistema digital que les seves sortides són funció exclusiva del valor de les seves entrades en un moment donat.*

[6] *Circuit Lògic Seqüencial: Sistema digital en que les seves sortides són funció del valor de les seves entrades i de l'estat anterior que emmagatzemen els seus components. Acostumen a estar governats per una senyal de rellotge.*

CAL_assembler

DESCRIPCIÓ

El processador *CAL16* executa diferents instruccions codificades en llenguatge màquina. El *CAL16* disposa d'un llenguatge assemblador, anomenat *CAL_assembler*, format per un conjunt d'instruccions reduït, amb les següents característiques:

- Existeixen 14 instruccions[1].
- Cada instrucció es codifica amb una tira de mida fixa de 16 bits.
- Existeixen 4 formats d'instruccions, en funció del nombre d'operands que necessiten.

FIGURA 54 – Formats bàsics de les instruccions

- Per a codificar les instruccions s'utilitza un codi d'operació de 4 bits, el que ens permetrà un màxim de 16 instruccions.
- Hi ha codis d'operació actualment lliures, és a dir sense cap instrucció assignada, amb l'objectiu de permetre la inserció de noves instruccions al conjunt actual.
- No es detecta overflow en l'execució de les instruccions.
- Només s'accepten immediats codificats en decimal pel llenguatge assemblador.
- Als immediats d'entrada se'ls estén el signe fins a 16 bits a partir del nombre de bits determinat pel tipus d'instrucció que es tracti.
- Els valors dels registres i dels Immediats es tractaran com a nombres enters.
- Els nombres enters estan codificats en *Complement a 2 (Ca2)*[2] amb 16 bits.
- En les instruccions, els immediats han d'estar codificats en decimal, en cas de tractar-se d'un nombre negatiu anirà precedit pel signe "-".
- Cal16 és una arquitectura amb *Seqüenciament Implícit*[3] mitjançant un registre especial anomenat *Program Counter*. Després de l'execució de cada instrucció aquest s'incrementa en 1 (PC=PC+1), menys en el cas de les instruccions de ruptura de seqüència (salts) que caldrà avaluar la condició.

---

[1] *14 Instruccions: Definides i descrites al capítol Annexes, Annex 4.*

[2] *Complement a 2 : Sistema de codificació de nombres enters en binari. Per més informació veure capítol Annexes, Annex 3.*

[3] *Seqüenciament Implícit : Forma d'autocàlcul en l'execució d'una instrucció, que calcula la direcció de la següent instrucció a executar pel processador sense necessitat d'una instrucció explicita que avanci l'estat del processador. Concretament en el processador CAL16 el seqüenciament implícit consta en incrementar en 1 la direcció de la instrucció actual.*

- Mitjançant combinacions de les instruccions actuals es poden arribar implementar altres funcionalitats de les que no se'n disposa d'instrucció (per exemple una Resta).
- Es poden utilitzar etiquetes per a determinar la destinació de les instruccions de salt.
- Les etiquetes segueixen la següent sintaxi:
    - **etiq_*nomEtiqueta* :** On < *nomEtiqueta* > es pot canviar per qualsevol nom que contingui qualsevol lletra o nombre.
- Existeixen dos tipus d'etiquetes:
    - **"etiq_*nomEtiqueta:*" :** Aquest tipus identifica la instrucció destí del salt. L'etiqueta ha de precedir la instrucció, ja sigui en la mateixa línia com en línies anteriors, sempre i quan no hi hagi cap altre instrucció enmig. Els ':' són importants ja que identifiquen que es tracta d'un punt destí de salt.
    - **"etiq_*nomEtiqueta*" :** Aquest tipus s'utilitza com a operand d'una instrucció. Si es tracta d'una instrucció de càrrega d'immediat, aquesta es sobreescriurà amb el valor absolut de la direcció on es troba la seva etiqueta associada (amb el mateix *nomEtiqueta*) de tipus destí del salt. En cas de tractar-se d'una operació de salt relatiu al comptador de programa (PC), l'etiqueta es reemplaçarà pel valor associat a la diferencia de posicions que hi ha entre la direcció en la que es troba l'etiqueta de destí de salt associada, menys, la direcció de la instrucció en la que es troba aquesta etiqueta.

INSTRUCCIONS

En la següent figura es pot veure la codificació i el format de les instruccions agrupades per tipologia. Els quatre bits de més pes sempre codifiquen el codi de l'operació, mentre que els següents depèn del tipus d'instrucció, poden codificar registres o immediats en funció dels seus operand.

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	
00XX	Ra	Rd	Rb	Operacions Aritmètiques i Lògiques [AND, OR, XOR, ADD]
010X	Ra	Rd	Imm-4b	Suma amb immediat i Desplaçament de bits [ADDI, ROTR]
0110	Ra	Rd	Imm-4b	Lectura de Memòria de Dades [LD]
0111	Ra	Rb	Imm-4b	Escriptura en Memòria de Dades [ST]
100X	Rd	Imm-8b		Càrrega d'Immediat [LI, LIH]
101X	Ra	Imm-8b		Ruptura de Seqüència Relativa [BNEG, BZ]
1100	Ra	Rd	Imm-4b	Ruptura de Seqüència amb link a l'estat actual [JR]
1101	Ra	00000000		Ruptura de Seqüència Absoluta [JMP]
1110	Imm-12b			Ruptura de Seqüència Absoluta [JMPI]

FIGURA 55 – Format de les instruccions

Per a l'anàlisi de cadascuna de les instruccions disponibles actualment al llenguatge màquina del *CAL16,* cal veure el capítol Annexes, l'Annex 4, on cada instrucció disposa de la seva descripció semàntica i funcional.

COMPILADOR

PROCÉS DE COMPILACIÓ

Per a portar a terme el procés de compilació del llenguatge *CAL_ assembler*, es parteix en quatre subprocessos. Cadascun d'aquests, a partir d'una entrada, té la responsabilitat d'analitzar les dades que li arriben, tractar-les i generar un tipus de sortida determinat. El següent diagrama mostra els passos a seguir, amb les entrades i sortides per cadascun.

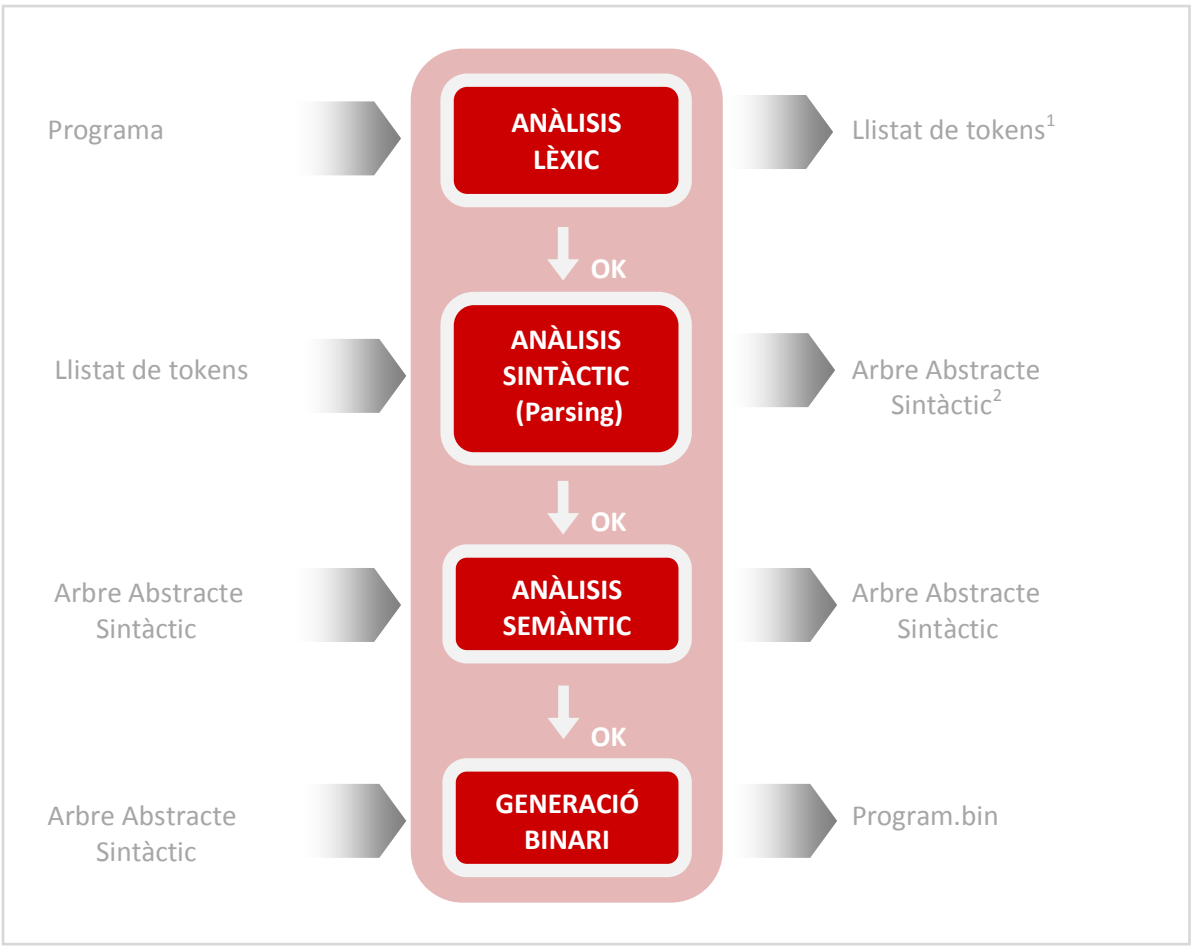


FIGURA 56 – Procés Compilació

[1] *Token : Cadena de caràcters amb un significat coherent dins d'un llenguatge concret.*

[2] *Arbre Sintàctic Abstracte (AST) : Tipus de dades que permet representar l'estructura sintàctica d'un codi font mitjançat un arbre. Per més informació veure capítol Annexes, Annex 5.*

• ANÀLISIS LÈXIC

És l'encarregat de preparar les dades pel Parser[3], elimina aquella informació no rellevant i parteix el programa d'entrada en un conjunt de tokens, que permetrà a l'anàlisi sintàctic detectar errors en la formació d'instruccions.

INPUT

Aquest procés rep com a entrada un fitxer que conté el programa en llenguatge assemblador, seguint les normes definides en aquest document per al nostre llenguatge.

OUTPUT

En aquest punt de la compilació es genera una llistat de tokens (agrupació de símbols identificats com a conjunt), els quals seran tractats en la següent fase pel Parser.

ERRORS

En aquesta fase es poden detectar errors com la presència de símbols que no formen part de llenguatge i ens permet encapsular informació com la dels registres, ja que en el nostre llenguatge hi fem referència com a Rx (on x=[0..15]) o com les etiquetes (etiq_nom).

• ANÀLISIS SINTÀCTIC

Aquest procés té com a objectiu crear un Arbre Sintàctic Abstracte (AST) partint de la seqüència de tokens generada pel procés anterior. Per aconseguir-ho, cal definir una gramàtica que descrigui el nostre llenguatge, i mirar si el llistat de tokens generat partint del nostre programa, és acceptat per aquesta gramàtica. En cas afirmatiu, ens indica que les instruccions del nostre programa estan ben formades.

INPUT

Aquest procés rep com entrada la seqüència de tokens generada pel procés anterior.

OUTPUT

Genera un Arbre Sintàctic Abstracte (AST), el qual té una arrel que simbolitza el programa i sota d'aquest pengen cadascuna de les instruccions que el formen. Alhora aquestes tenen com a fills els seus operands, ja siguin registres o immediats. Per a cada component s'emmagatzema en el node el tipus (kind) i la informació (text).

ERRORS

En aquest punt de compilació podem detectar errors com una instrucció mal formada, és a dir, que no segueix el patró de tipus d'instruccions definides.

[3] *Parser : Analitzador sintàctic que forma part d'un compilador encarregat de transformar l'entrada generada com a un llistat de tokens cap a un arbre de derivació, com pot ser un AST.*

• ANÀLISIS SEMÀNTIC

Per a realitzar l'anàlisi semàntic és necessari recórrer l'arbre AST que arriba. Aquest procés té com a objectiu realitzar comprovacions com instruccions ben formades, immediats dins el rang o etiquetes ben associades. Partint d'un Arbre Sintàctic Abstracte (AST) que ens proporciona l'etapa anterior, primer de tot es crea l'estructura de dades de *labtab* (una taula que emmagatzema per a cada etiqueta destí de salt quina és la seva direcció associada), aquesta s'inicialitza amb la primera passada a l'arbre.

Un cop totes les etiquetes estan registrades, s'inicia la segona passada, en la que es modifica l'ordre dels operands de les instruccions en funció del tipus de instrucció, es comprova que totes estiguin ben formades, es controlen els rangs del immediats, i es sobreescriuen els operands de les instruccions que corresponen a etiquetes, pel seu valor corresponent:

- **BZ ó BNEG:** Es calcula la diferencia entre la direcció destí de salt i la direcció actual, per aconseguir l'immediat associat a aquest salt, que en el moment d'execució es tracta com un sal relatiu al PC.
- **LI, LHI, JMPI:** Es reemplaça l'etiqueta de l'operand amb la seva direcció destí associada, que es troba emmagatzemada a l'estructura de dades *labtab.* En el cas de LI, s'agafen els 8 bits de menys pes, en el cas de LIH els 8 de més pes i en el cas de JMPI directament el seu valor.

INPUT

Aquest procés rep com entrada l'AST generat pel procés anterior.

OUTPUT

Retorna el mateix Arbre Sintàctic Abstracte(AST), rebut, però ara modificat i decorat en funció de les instruccions que formen el programa que representa, amb l'objectiu de facilitar la feina a la generació de codi.

ERRORS

En aquest punt de compilació podem detectar errors com una instrucció concreta mal formada, és a dir, per exemple un ADDI sense immediat. També podem detectar una mala correspondència, entre etiquetes operands i etiquetes destí de salt o fins i tot, immediats no representables amb el nombre de bits que tenen associats.

• GENERACIÓ DEL BINARI

Aquesta fase rep el AST generat per la fase anterior i se'l recórrer instrucció a instrucció per a crear un fitxer on escriu cadascuna de les instruccions codificades en binari. Per a cada instrucció primer escriu el binari relatiu al seu codi d'operació, i li concatena la representació binaria dels registres i immediats que la formen, tots aquest ja ordenats convenientment en el procés anterior.

INPUT

Aquest procés rep com entrada l'AST generat pel procés anterior.

OUTPUT

Crea un fitxer *Program.bin* amb la representació binaria del conjunt d'instruccions que formen el programa. En aquest trobem un seguit de tires de bits separades per salts de línea, cada línea és la representació d'una instrucció, se'n manté l'ordre. Aquest fitxer el col·loca automàticament el compilador al directori d'on es llegeixen els valors inicials de les memòries. En aquest cas aquest fitxer serà el que es carregarà a la memòria d'instruccions per executar el programa.

ERRORS

Aquest punt no serveix per a detectar errors, sinó que el fet d'arribar-hi ja significa que el programa és correcte en tots els sentits. Aquest procés es limita a passar a binari la informació rebuda i crear una fitxer que per a emmagatzemar aquestes tires de bits.

ENTORN, EINES I LLENGUATGE DE DESENVOLUPAMENT

L'entorn en el que s'executa el compilador és sota la plataforma del sistema operatiu de Windows, on es fa córrer l'eina *PCCTS*[4] basada en 2 passos bàsics: la construcció d'un analitzador(parser) descendent[5], que permet la generació i el recorregut per realitzar comprovacions de Arbres Abstractes Sintàctics (AST) , a partir dels quals es genera el programa binari.

Aquest passos es porten a terme mitjançant la utilització del programa *ANTLR*[6] que s'encarrega de la generació d'un fitxer programat en C, *cl.c, que* guia el procés de compilació. La seva generació es porta a terme a partir del fitxer de definició *cl.g,* que conté la gramàtica i les tasques de comprovació a portar a terme, mitjançant el recorregut del AST; aquest procés comença en la generació de l'arbre AST un cop el programa és reconegut per la gramàtica, passa per comprovacions de tipus semàntic que es realitzen en diversos recorreguts de l'arbre generat, i finalitza en la generació del fitxer binari que inclou el programa assemblat.

Veiem ara més concretament cadascun d'aquests processos que s'executen gràcies a les eines utilitzades:

- **DESCRIPCIÓ**

El primer pas que realitza *ANTLR,* és a partir d'una gramàtica que defineix el llenguatge assemblador del *CAL16* (descrita en el fitxer cl.g), generar una reconeixedor el llenguatge definit per la gramàtica descrita, que consta d'un escàner[7] (que es troba en el fitxer generat scan.c) i d'un parser (que es troba en el fitxer generat cl.c) encarregat de comprovar que es compleix la gramàtica.

---

[4] *PCCTS : Purdue Compiler Construction Tool Set, projecte encarregat de la generació de parser a partir de la definició d'una gramàtica.*

[5] *Parser d'Anàlisis Descendent : Parser que recórrer l'arbre AST de l'arrel cap a les fulles.*

[6] *ANTLR : Another Tool for Language Recognition, eina que opera sobre llenguatges, proporcionant un marc per a construir parsers, intèrprets, compiladors i traductors de llenguatges a partir de les seves descripcions gramaticals. Per més informació veure capítol Annexes, Annex 6.*

[7] *Escàner : Primera etapa del procés de compilació, concretament la d'anàlisis lèxic, basat en una màquina d'estats finits encarregat de reconèixer els tokens.*

Per aconseguir-ho, el fitxer parser.dlg (que conté la definició de l'escàner), rep un programa escrit en assemblador d'entrada i li aplica un Anàlisis Lèxic, el qual s'encarregarà d'identificar un llistat de tokens definits.

```
#lexclass START
#token OP            "\("
#token CP            "\)"
#token COMA          ","
#token REG          "[rR][0-9]+"
#token LABELPOINT   "etiq_[a-zA-Z0-9]*:"
#token LABEL        "etiq_[a-zA-Z0-9]*"
#token INST         "[a-zA-Z][a-zA-Z]*"
#token IMM          "[0-9]+"
#token NEG          "\-"
#token COMMENT       "//~[\n]*"  << printf(zzlextext); zzskip(); >>
#token WHITESPACE    "[\ \t]+"   << printf("%s",zzlextext); zzskip(); >>
#token NEWLINE       "\n"        << zzline++; printf("\n%3d: ", zzline); zzskip(); >>
#token LEXICALERROR "~[]"        << printf("\nLexical error: symbol '%s' ignored!\n",
                                     zzlextext); zLexErrCount++; zzskip(); >>
#token INPUTEND      "@"
```

FIGURA 57 –Llistat de tokens reconeguts

Quan el llistat de tokens està generat el passa com a entrada al primer dels anàlisis, l'Anàlisi Sintàctic, que permet assegurar que aquest llistat és reconegut per la gramàtica definida.

```
program: ll_instr INPUTEND!  <<createASTlist>>;

ll_instr: (instr | LABELPOINT)* ;

instr: INST^ cos;

cos: (  immediat (OP! REG CP! COMA! REG |  )
      | REG ( COMA! (  REG COMA! ( REG | immediat) | immediat ( OP! REG CP! |  ) )| ));

immediat:  (( NEG^ | ) IMM | LABEL);
```

FIGURA 58 –Gramàtica de descripció del llenguatge CAL_assembler

Durant aquest procés, es determina la generació d'un error per cada entrada no reconeguda, és a dir, si no es reconeix el símbol utilitzat en algun token o bé si la seqüència d'entrada no s'ajusta a l'especificada per la gramàtica. En cas de no trobar errors durant aquest procés, es genera un AST que representarà el programa a compilar en forma d'arbre on l'arrel és el conjunt d'instruccions i hi ha un fill per cada instrucció i etiqueta destinació de salt, que forma part del programa, i on per cada instrucció alhora, s'encapsulen els seus operands com a fills seus, enregistrant alhora el seu tipus.

A partir d'aquest punt comença el segon procés d'anàlisis, l'Anàlisi Semàntic, descrit també en el fitxer cl.c, que inicialment mitjançant un recorregut descendent de l'arbre, enregistra en una estructura de dades totes aquelles etiquetes destinació de salts juntament amb l'adreça a la que fan referència, i amb una segona passada realitza comprovacions de tipus d'operands sobre les diferents instruccions. En cas de no trobar errors, s'inicia l'última passada descendent per l'arbre que crea el fitxer binari resultant, on per cada instrucció aquesta es assemblada juntament amb els seus operands en una la tira de 16 bits.

- **JUSTIFICACIÓ**

Aquest compilador està basat en el compilador explicat a l'assignatura impartida a la FIB en l'assignatura de Compiladors i això justifica l'ús de les eines escollides. L'únic aspecte que no he mantingut ha estat el sistema operatiu, ja que en l'assignatura s'utilitza el sistema operatiu de Linux, però el fet de que el processador i la ISE de Xilinx que se'm va facilitar des d'un principi funcionava sobre Windows, vaig creure oportú el fet que, tant el compilador, com el simulador i l'eina d'execució poguessin córrer sobre el mateix sistema operatiu, fet que aporta comoditat i eficiència pel desenvolupament de nous programes i l'execució d'aquests.

Es podrien haver utilitzat altres entorns o eines per a desenvolupar-lo, però al tractar-se d'un petit apartat del projecte, i no de l'objectiu central, no quedava dins l'abast definit l'estudi d'altres possibilitats en la seva implementació.

Crec que en aquest projecte és útil disposar d'aquest compilador per a fer el processador més manejable, ja que no és el mateix haver de programar en binari que poder-ho fer en assemblador. Ja que l'objectiu principal del projecte és realitzar-ne un ús docent, el compilador és útil de cara a la manipulació del llenguatge assemblador per part dels alumnes, per a detectar errors en la programació i generar el programa en binari. Aquest fet permet incloure exercicis de generació de programes en llenguatge assemblador que un cop compilats es puguin provar d'executar en el propi processador.

EXEMPLES

A continuació es mostra amb diferents exemples la compilació de diversos programes escrits en *CAL_assembler*. Cadascun d'ells inclou alguns punts típics d'error. Finalment l'últim exemple es tracta d'un programa correcte del qual es pot veure el fitxer binari resultant.

• ERROR LÈXIC

El següent programa conté diferents símbols que el compilador no té definits en el seu llistat de tokens i que per tant no pot reconèixer. Quan hi ha un problema d'aquest tipus el compilador respon identificant els errors de la forma que mostra la següent figura:

```
etiq_main:
    LIH R1,1
    LI R2,
    LD R3,(R1)5
inf:
    ST 0 R0,R3
    ADD R4,R2,R3
    XOR R4,4,R8
    ADDI R4,4,1
```

```
  1: etiq_main:
  2:     LIH R1,1
  3:     LI R2,
Lexical error: symbol '<' ignored!
6
  4:     LD R3,5(R1)
  5: inf
Lexical error: symbol ':' ignored!

  6:     line 6: syntax error at "ST" missing ( REG LABEL IMM N
ST 0(R0),R3
  7:     ADD R4,R2,R3
  8:     XOR R4,R4,R8
  9:     ADDI R4,R4,1
 10:
There were lexical errors.
```

FIGURA 59 –Exemple Error Lèxic

• ERROR SINTÀCTIC

Un error sintàctic es dóna quan el programa no compleix el format de les instruccions definit per la gramàtica. El següent programa conté diferents instruccions mal formades, però el compilador para el seu procés de control al trobar el primer error que fa que no sigui reconegut, és per aquest motiu que tant sols es mostra el primer dels errors, en cas de que aquest fos solvatat al tornar a compilar es detectaria el següent.

```
etiq_main:
    LIH R1,1
    LI R2,
    LD R3,5(R1)
etiq_inf:
    ST 0(R0),R3
    OR R4,R4,R8
    ADDI R4,4,R1
```

```
  1: etiq_main:
  2:     LIH R1,1
  3:     LI R2,
  4:     line 4: syntax error at "LD" missing
         ( REG LABEL IMM NEG ) LD R3,5(R1)
  5: etiq_inf:
  6:     ST 0(R0),R3
  7:     ADD
  8:     line 8: syntax error at "XOR" missing
         ( REG LABEL IMM NEG )
line 8: syntax error at "R4" missing INPUTEND
  9:
 10:
There were syntax errors.
```

FIGURA 60 –Exemple Error Sintàctic

- **ERROR SEMÀNTIC**

Un cop el programa ja no conté ni errors lèxics ni errors sintàctics, el compilador genera l'arbre abstracte sintàctic que correspon al codi i llavors el recórrer per a comprovar si es compleix la semàntica. En aquest procés es detecten errors com immediats fora de rang o si una instrucció en concret, tal i com es mostra a continuació:

```
etiq_main:
    LIH R1,1
    LI R2,1024
    LD R3,5(R1)
etiq_inf:
    ST 0(R0),R3
    OR R4,R4,R8
    ADDI R4,R4,R1
```

```
 1: etiq_main:
 2:     LIH R1,1
 3:     LI R2,1024
 4:     LD R3,5(R1)
 5: etiq_inf:
 6:     ST 0(R0),R3
 7:     ADD R4,R4,R8
 8:     XOR R4,R4,R8
 9:     ADDI R4,R4,R1
10:

list
¦ LabelPoint(etiq_main:)
¦ Instruction(lih)
¦ ¦ Register(1)
¦ ¦ Offset(1)
¦ Instruction(li)
¦ ¦ Register(2)
¦ ¦ Offset(1024)
¦ Instruction(ld)
¦ ¦ Register(3)
¦ ¦ Offset(5)
¦ ¦ Register(1)
¦ LabelPoint(etiq_inf:)
¦ Instruction(st)
¦ ¦ Offset(0)
¦ ¦ Register(0)
¦ ¦ Register(3)
¦ Instruction(add)
¦ ¦ Register(4)
¦ ¦ Register(4)
¦ ¦ Register(8)
¦ Instruction(xor)
¦ ¦ Register(4)
¦ ¦ Register(4)
¦ ¦ Register(8)
¦ Instruction(addi)
¦ ¦ Register(4)
¦ ¦ Register(4)
¦ ¦ Register(1)

Parts Checking:
L. 3: Offset 1024 out of range [ -128...127 ].
L. 9Instruction addi not match, expected Offset found Register.

There are errors: no code generated
```

FIGURA 61 –Exemple Error Semàntic

• GENERACIÓ D'UN BINARI

Quan el programa ja no conté cap tipus d'error, el compilador genera una tira de 16 bits corresponen a cada instrucció i les escriu en un fitxer binari anomenat **program.bin** que col·loca al directori **Processador\Programes**. Aquests fitxer és el que s'utilitzarà en l'execució del programa per a inicialitzar la Memòria d'Instruccions del processador CAL16. El següent codi ja no conté errors i la imatge mostra la traducció del programa escrit en CAL_assembler a binari:

```
etiq_main:
    LIH R1,1
    LI R2,1024
    LD R3,5(R1)
etiq_inf:
    ST 0(R0),R3
    OR R4,R4,R8
    ADDI R4,R4,R1
```

```
 1: etiq_main:
 2:     LIH R1,1
 3:     LI R2,8
 4:     LD R3,5(R1)
 5: etiq_inf:
 6:     ST 0(R0),R3
 7:     ADD R4,R4,R8
 8:     XOR R4,R4,R8
 9:     ADDI R4,R4,1
10:

list
| LabelPoint(etiq_main:)
| Instruction(lih)
| | Register(1)
| | Offset(1)
| Instruction(li)
| | Register(2)
| | Offset(8)
| Instruction(ld)
| | Register(3)
| | Offset(5)
| | Register(1)
| LabelPoint(etiq_inf:)
| Instruction(st)
| | Offset(0)
| | Register(0)
| | Register(3)
| Instruction(add)
| | Register(4)
| | Register(4)
| | Register(8)
| Instruction(xor)
| | Register(4)
| | Register(4)
| | Register(8)
| Instruction(addi)
| | Register(4)
| | Register(4)
| | Offset(1)

Parts Checking:
   OK!

Generating Binary...
Binary Generated!!
```

```
1001000100000001
1000001000001000
0110000100110101
0111000000110000
0011010001001000
0010010001001000
0100010001000001
0000000000000000
0000000000000000
```

FIGURA 62 –Exemple sense errors

INTEGRACIÓ i PROVES

ENTORN I EINES

Per a portar a terme les proves del Processador *CAL16*, s'utilitza l'entorn de Xilinx ISE Design Tools 12.1, el qual ens aporta una eina de simulació d'execució i una de sintetització del codi, i programació d'una FPGA, més concretament la FPGA utilitzada és una Spartan 3E. S'utilitza el sistema operatiu de Windows.

L'eina de simulació permet executar el processador, que llegeix el programa d'entrada i executa segons el seu comportament. Com a sortida genera un cronograma on es poden consultar el valor de tots els busos del processador, on en funció del programa es mostra en cada exemple aquells busos més significatius per a mostrar el comportament del *CAL16*.

L'eina de programació de la FPGA permet mapejar el processador *CAL16* dins la FPGA per que aquesta es comporti com ho faria el processador. Els exemples escollits per aquest tipus d'execució són exemples que interactuen amb l'espai d'entrada sortida del processador, que estan directament connectats als dispositius de interruptors i leds de la FPGA.

A més per a la comprovació de la correctesa dels programes escrits en *CAL_assembler* i la generació del fitxer binari que es carrega a la Memòria d'Instruccions per a comprovar el correcte funcionament del processador, es fa ús del compilador explicat en l'apartat anterior.

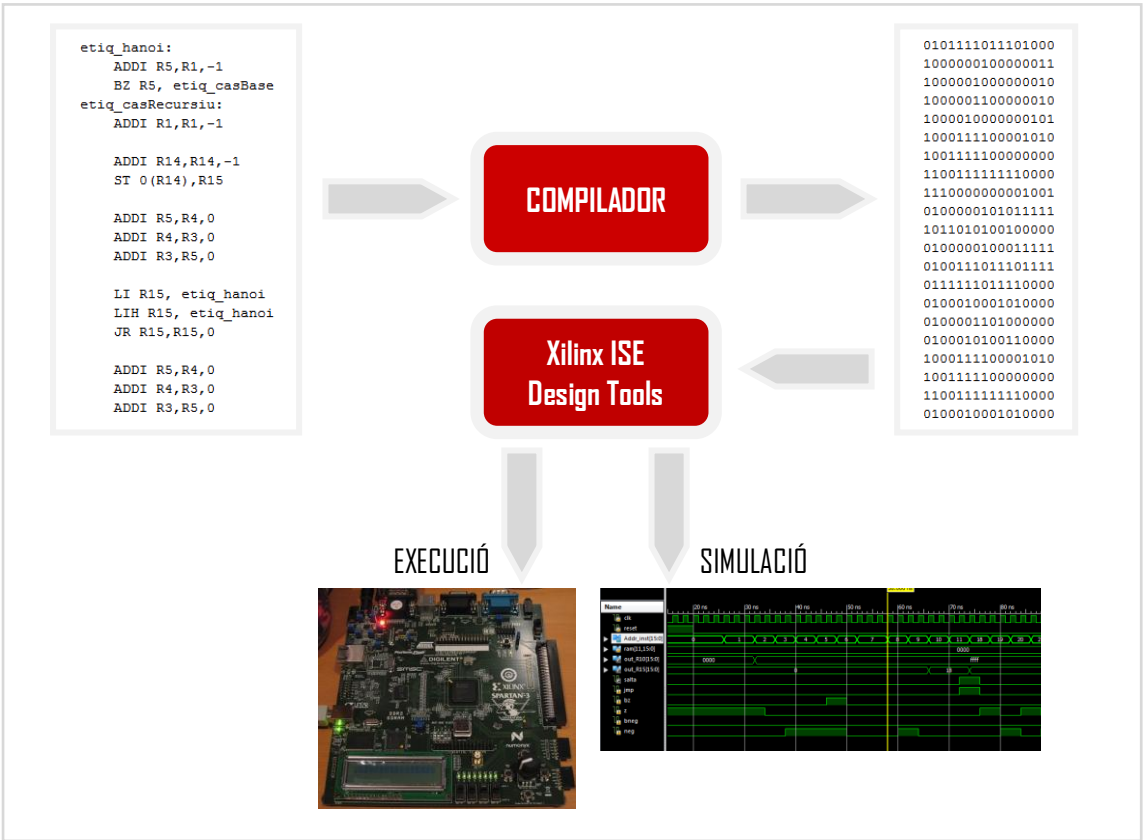


FIGURA 63 – Procés d'evolució d'un programa en CAL_assembler per obtindre la seva execució.

SIMULACIÓ

En aquest apartat es presenten diverses proves realitzades amb el *CAL16*. Els programes que s'executen tenen la característica de que no interactuen amb els dispositius d'entrada/sortida, ja que aquest és un punt d'anàlisis en el següent apartat.

Per a cada exemple hi ha una primera explicació de quin bloc del processador es objectiu de test, després es presenta el codi en *CAL_assembler* que s'executa en el processador i que és l'encarregat de testejar la part escollida, i finalment mitjançant un cronograma d'execució del programa que genera l'eina de simulació, es mostren aquelles senyals i busos escollits que permeten observar el correcte funcionament del processador.

Exemple 1: Funció màxim d'enters

OBJECTIU: Amb aquest exemple es vol testejar les operacions aritmètiques i lògiques que formen el bloc ALU del *CAL16*.

{ Pre: Els nombres a comparar són enters i es troben en les adreces 0x00 i 0x01 de la Memòria de Dades respectivament. }

{ Post: R0 conté el màxim entre els dos nombres. }

```
R1 = MemData[0x00];
R2 = MemData[0x01];

if( (R1[15] ^ R2[15]) != 0){ //Si signes diferents
    if(R2 >= 0) R0 = R2
    else R0 = R1

}else if( R2 != 0x8000 ){   //Si R2 no és el mínim nombre representable
    if((R1 - R2) >= 0){
        R0 = R1
    }else{
        R0 = R2
    }
}else{                      //Si R2 és el mínim nombre representable
    R0  = R1
}
```

FIGURA 64 – Codi en C del programa Funció Màxim d'Enters.

```
0:   LI R0,0
1:   LD R1,0(R0)     //Llegir nombre
2:   LD R2,1(R0)     //Llegir nombre
3:   LI R3,0
4:   LIH R3,-128
5:   AND R0,R2,R3    // Agafar el signe de R2
6:   AND R3,R1,R3    // Agafar el signe de R1
7:   XOR R3,R0,R3
8:   BZ R3, etiq_else0     //Salta si signes diferents
9:   BNEG R2, etiq_else3 //Salta si R2 és negatiu
10:  OR R0,R2,R2     // max = R2
11:  JMPI etiq_fi
etiq_else3:
12:  OR R0,R1,R1     // max = R1
13:  JMPI etiq_fi
etiq_else0:
14:  LI R3,0
15:  LIH R3,-128
16:  XOR R3,R2,R3
17:  BZ R3, etiq_else1     //Salta si R2 = 0x8000
18:  LI R3,-1
19:  XOR R3,R2,R3
20:  ADDI R3,R3,1     // R3 = – R2
21:  ADD R3,R1,R3     // R1 – R2
22:  BNEG R3, etiq_else2     //Salta si R2 > R1
23:  ADDI R0,R1,0     // max = R1
24:  JMPI etiq_fi
etiq_else2:
25:  ADDI R0,R2,0     // max = R2
26:  JMPI etiq_fi
etiq_else1:
27:  ADDI R0,R1,0     // max = R1
etiq_fi:
28:  JMPI etiq_fi
```

FIGURA 65 – Codi en CAL_assembler del programa Funció Màxim d'Enters.

A continuació es presenten un seguit de simulacions per a demostrar el correcte funcionament del programa executat amb el *CAL16*. Amb la primera simulació es realitza un anàlisis en detall de diversos punts que són objectiu d'estudi mitjançant un cronograma complet. En les següents simulacions tant sols es mostra un cronograma simplificat per demostrar el correcte funcionament.

EL valor de les senyals i busos del *CAL16* que es mostren se'ls fa referència amb la següent nomenclatura:

clk : Senyal de rellotge que marca la velocitat del processador.

reset : Senyal de reinici d'execució.

Addr_inst [15:0] : Adreça en decimal referent a la instrucció a executar, valor del registre PC.

out_Rx [15:0] : Valor en hexadecimal del registre Rx, on x equival al nombre del registre, en aquest cas es mostren tant sols els que utilitza el programa (x = {0,1,2,3}).

A [15:0] i B [15:0] : Valors en hexadecimal dels busos d'entrada al bloc ALU.

op [3:0] : Valor en binari del bus d'entrada de control d'operació del bloc ALU.

W [15:0] : Valor en hexadecimal del bus de sortida del bloc ALU per on es serveix el resultat de l'operació.

neg : Senyal de sortida del bloc ALU que s'activa quan el resultat de l'operació, que es serveix pel bus W, és negatiu.

z : Senyal de sortida del bloc ALU que s'activa quan el resultat de l'operació, que es serveix pel bus W, té el valor de 0.

fetch : Senyal de control del processador generada per la Unitat de Control, que identifica l'estat en el que es carrega la nova instrucció a executar de la Memòria d'Instruccions.

exec : Senyal de control del processador generada per la Unitat de Control, que identifica l'estat en el que es descodifica i s'executa la instrucció actual.

mem : Senyal de control del processador generada per la Unitat de Control, que identifica l'estat en el que es realitza l'accés a Memòria de Dades, en cas que la instrucció ho requereixi.

- **Simulació 1:** Dos nombres positius MemData[0x00] = 0x595D
  MemData[0x01] = 0x595E

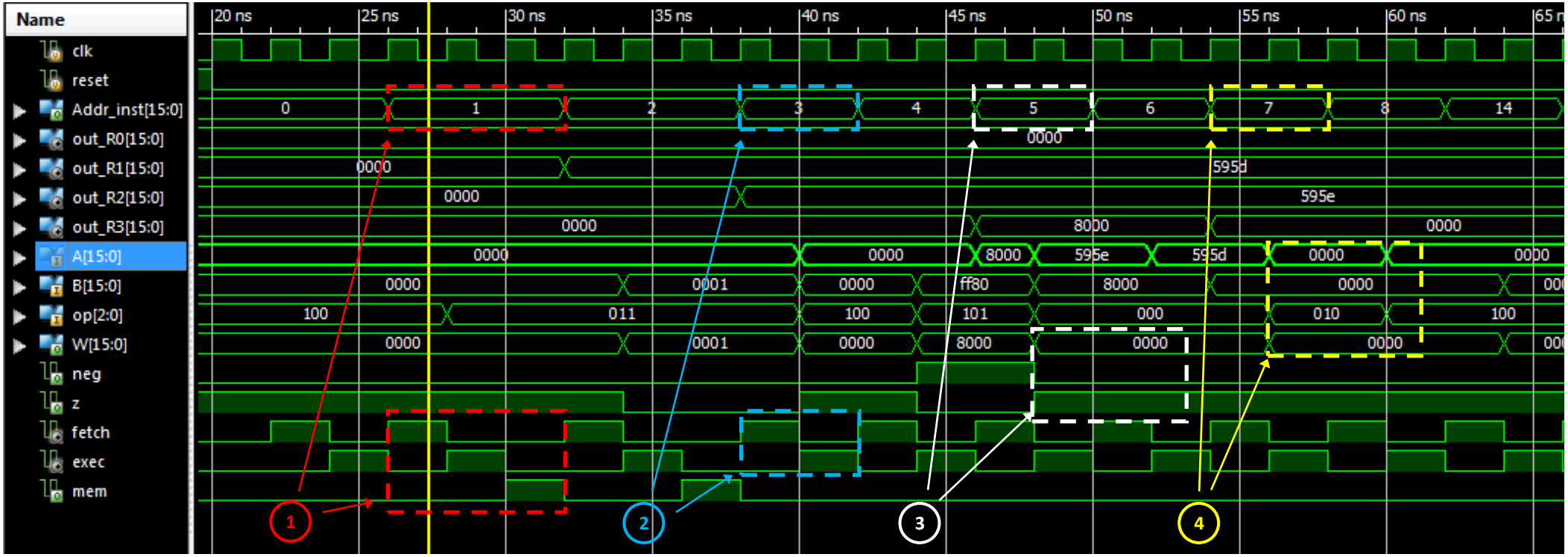


FIGURA 66 – Simulació la del programa Funció Màxim d'Enters.

1 – Adreça MI = 1, Instrucció = LD R1, 0(R1) Instrucció d'accés a MD on es pot veure les 3 etapes que executa la màquina d'estats fetch, exec i mem.

2 – Adreça MI = 3, Instrucció = LI R3, 0 Instrucció simple sense accés a MD on es pot veure les 2 etapes que executa la màquina d'estats fetch i exec.

3 – Adreça MI = 5, Instrucció = AND R0,R2,R3 Instrucció que genera el resultat 0x0000 en el bloc ALU i per aquest motiu s'activa la senyal z = 1.

4 – Adreça MI = 7, Instrucció = XOR R3,R0,R3 Instrucció XOR amb op = 100, que realitza aquesta operació sobre els signes dels operands 0x0000 ^ 0x0000 = 0x0000.

- **Continuació Simulació 1:** Dos nombres positius

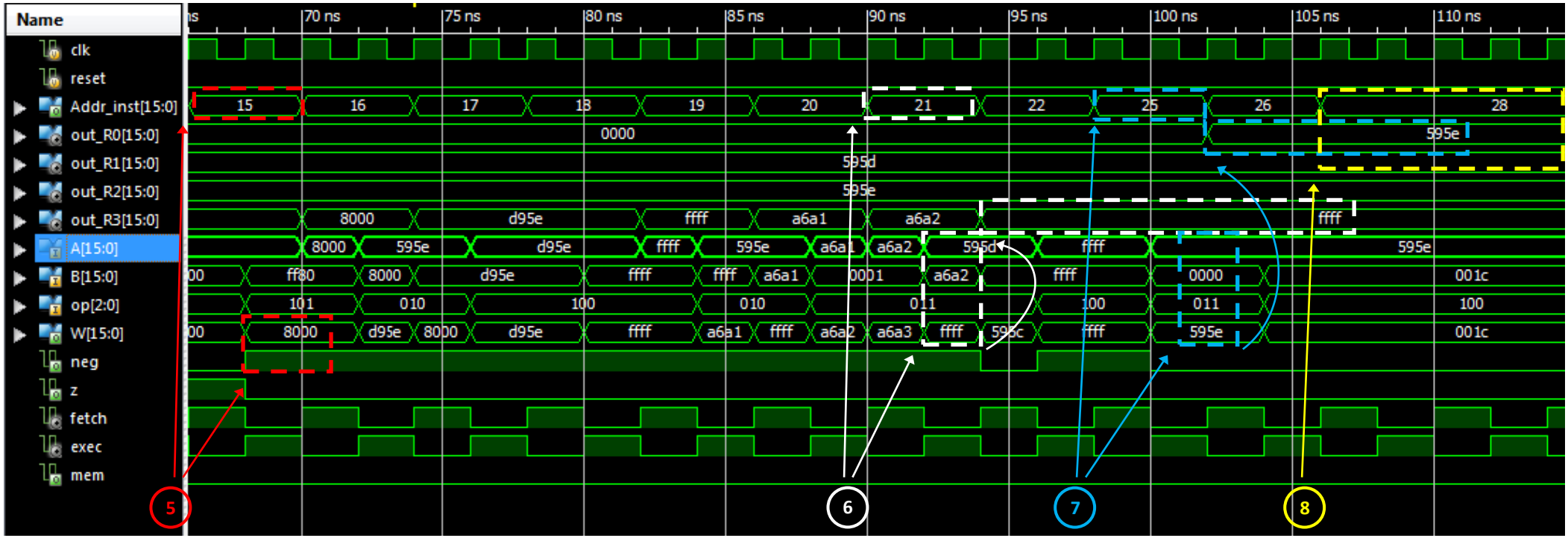


FIGURA 67 – Simulació 1b del programa Funció Màxim.

5 – Adreça MI = 15, Instrucció = LIH R3,-128 — Instrucció que genera el resultat 0x8000 en el bloc ALU i al ser un nombre negatiu activa la senyal neg = 1.

6 – Adreça MI = 21, Instrucció = ADD R3,R1,R3 — Instrucció de suma de registres, op = 011, on 0x595D + 0xA6A2 = 0xFFFF i aquest valor s'escriu al final d'etapa en R3.

7 – Adreça MI = 25, Instrucció = ADDI R3,R2,0 — Instrucció de suma amb immediat, op = 011, on 0x595E + 0x0000 = 0x595E i aquest valor s'escriu al final d'etapa en R0. El fet de sumar un immediat igual a 0 a un registre té com a objectiu realitzar un còpia del valor d'aquest registre a un altre.

8 – Adreça MI = 28, Instrucció = JMPI etiq_fi — Instrucció de final d'execució on veiem que el resultat després de l'execució emmagatzemat en el registre R0 és 0x595E.

- **Simulació 2:** Dos nombres negatius

  MemData[0x00] = 0xFFFF
  MemData[0x01] = 0x90D2

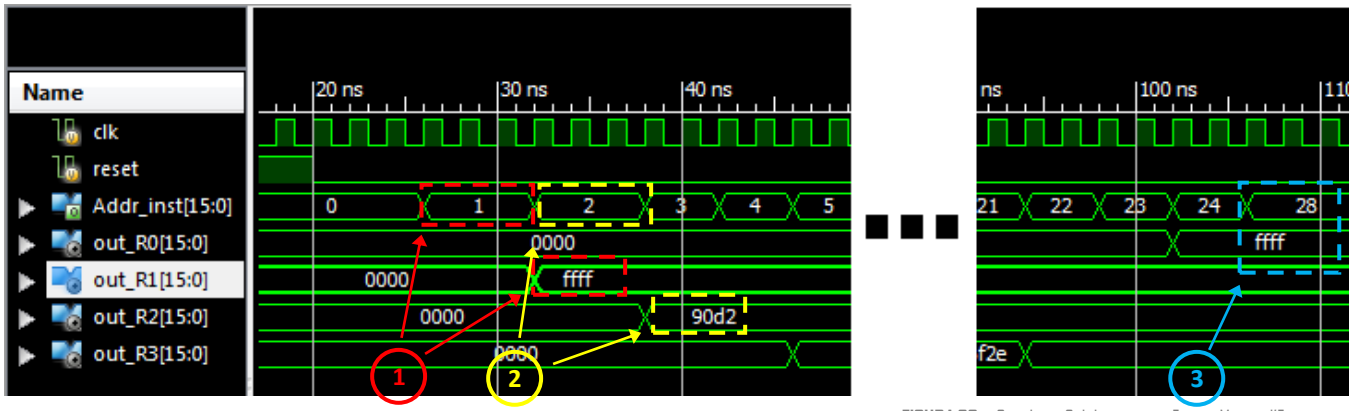


FIGURA 68 – Simulació 2 del programa Funció Màxim d'Enters.

1 – Adreça MI = 1, Instrucció = LD R1,0(R0) Càrrega del primer operand R1 = 0xFFFF.

2 – Adreça MI = 1, Instrucció = LD R2,1(R0) Càrrega del primer operand R2 = 0x902D.

3 – Adreça MI = 28, Instrucció = JMPI etiq_fi Final d'execució el resultat és R0 = 0xFFFF.

- **Simulació 3:** El nombre més petit representable

  MemData[0x00] = 0x0066
  MemData[0x01] = 0x8000

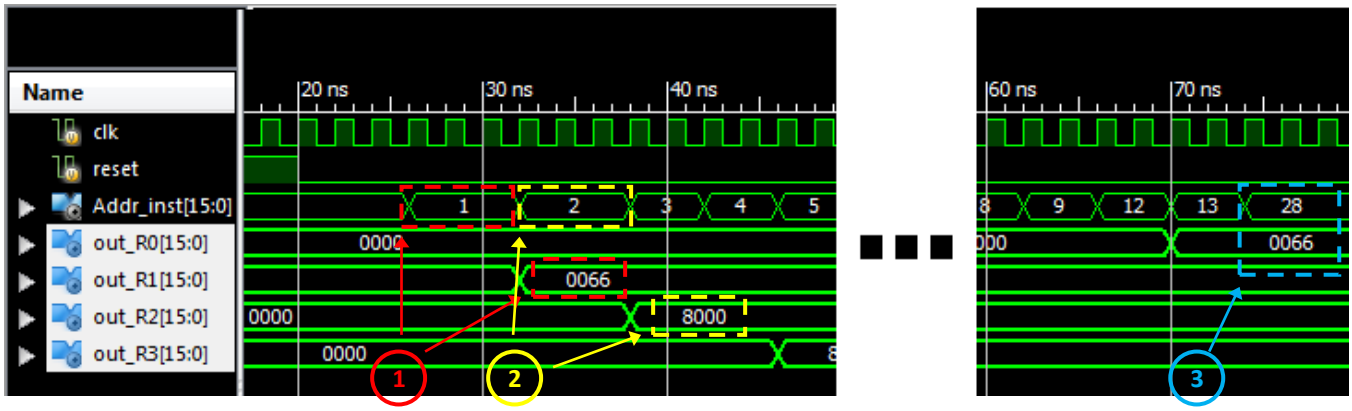


FIGURA 69 – Simulació 3 del programa Funció Màxim d'Enters.

1 – Adreça MI = 1, Instrucció = LD R1,0(R0) Càrrega del primer operand R1 = 0x0066.

2 – Adreça MI = 1, Instrucció = LD R2,1(R0) Càrrega del primer operand R2 = 0x8000.

3 – Adreça MI = 28, Instrucció = JMPI etiq_fi Final d'execució el resultat és R0 = 0x0066.

- **Simulació 4:** Signes diferents I

  MemData[0x00] = 0xC004
  MemData[0x01] = 0x0123

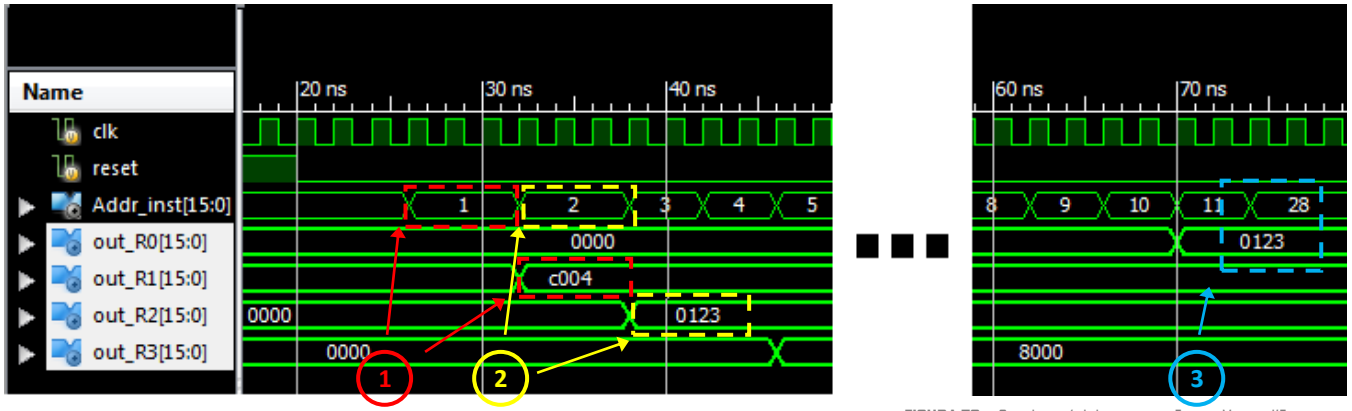


FIGURA 70 – Simulació 4 del programa Funció Màxim d'Enters.

1 – Adreça MI = 1, Instrucció = LD R1,0(R0) Càrrega del primer operand R1 = 0xC004.

2 – Adreça MI = 1, Instrucció = LD R2,1(R0) Càrrega del primer operand R2 = 0x0123.

3 – Adreça MI = 28, Instrucció = JMPI etiq_fi Final d'execució el resultat és R0 = 0x0123.

- **Simulació 5:** Signes diferents II

  MemData[0x00] = 0x8004
  MemData[0x01] = 0x0004

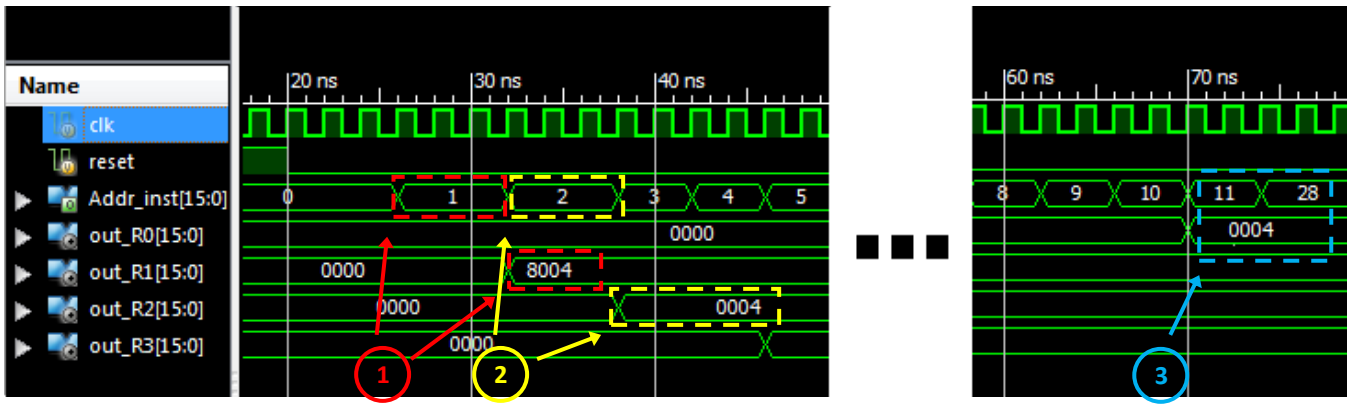


FIGURA 71 – Simulació 5 del programa Funció Màxim d'Enters.

1 – Adreça MI = 1, Instrucció = LD R1,0(R0) Càrrega del primer operand R1 = 0x8004.

2 – Adreça MI = 1, Instrucció = LD R2,1(R0) Càrrega del primer operand R2 = 0x0004.

3 – Adreça MI = 28, Instrucció = JMPI etiq_fi Final d'execució el resultat és R0 = 0x0004.

Exemple 2: Cerca màxim d'un vector

OBJECTIU: Amb aquest exemple es vol mostrar la crida a una funció auxiliar i el valor del registre PC en funció de les instruccions de ruptura de seqüència. Amb aquest objectiu es modifica l'exercici anterior per a que funcioni com una subrutina la qual serà invocada des d'un programa principal, que l'utilitzarà per cercar el màxim d'un vector.

Per implementar la crida a la funció auxiliar s'utilitza els registres R1 i R2 pel pas de paràmetres, el registre R0 pel retorn del resultat de la funció i R15 per a linkar la direcció de retorn de la crida.

{ Pre: El vector és de 10 enters i s'emmagatzema a partir de l'adreça 0x00 de Memòria de Dades. }

{ Post: La posició 0x0B emmagatzema el màxim valor del vector. }

```
int maxim (int a, int b){ // Els operands arriben per R1 i R2 respectivament
    if( (a[15] ^ b[15]) != 0){  //Si signes diferents
        if(b >= 0) max = b;
        else max = a;

    }else if( b != 0x8000 ){    //Si R2 no és el mínim nombre representable
        if((a - b) >= 0){
            max = a;
        }else{
            max = b;
        }
    }else{                      //Si R2 és el mínim nombre representable
        max = a;
    }
    return max;     //El retorn es deixa en R0
}

int main(){
    max = vector[0];
    for( int i = 1; i < 10; i++ ){
        max = maxim(max,vector[i]);
    }
    MemData[0x0B] = max;
}
```

FIGURA 72 – Codi en C del programa Cerca Màxim d'un Vector.

```
0:  LI R4,0          //R4 = &vec
1:  LD R10,0(R4)     //R10 = max
2:  LI R5,1          //R5 = I = 1
etiq_for:
3:  LI R6,-10
4:  ADD R6,R5,R6
5:  BZ R6,etiq_fifor
6:  ADD R6,R4,R5     //&vector[i]
7:  LD R2,0(R6)      //pas de paràmetre
8:  ADDI R1,R10,0    // pas de paràmetre
9:  LI R15,etiq_maxim
10: LIH R15, etiq_maxim
11: JR R15,R15,0     //Salta a la funció maxim
                     // R15 = link retorn
12: ADDI R10,R0,0    //max = retorn
13: ADDI R5,R5,1     //i++
14: JMPI etiq_for
etiq_fifor:
15: LI R6,11
16: ST 0(R6),R10     //memData[0X0B] = max
etiq_final:
17: JMPI etiq_final

etiq_maxim:          //R1 i R2 nombres a comparar
                     //R15 = Adreça de retorn
18: LI R3,0
19: LIH R3,-128
20: AND R0,R2,R3
21: AND R3,R1,R3
22: XOR R3,R0,R3
23: BZ R3, etiq_else0
24: BNEG R2, etiq_else3
25: OR R0,R2,R2
26: JMPI etiq_fi
etiq_else3
27: OR R0,R1,R1
28: JMPI etiq_fi
etiq_else0:
29: LI R3,0
30: LIH R3,-128
31: XOR R3,R2,R3
32: BZ R3, ETIQ_ELSE1

33: LI R3,-1
34: XOR R3,R2,R3
35: ADDI R3,R3,1
36: ADDI R3,R1,R3
37: BNEG R3, etiq_else2
38: ADDI R0,R1,0
39: jmpi etiq_fi
etiq_else2:
40: addi r0,r2,0
41: jmpi etiq_fi

etiq_else1:
42: addi r0,r1,0
etiq_fi:
43: jmp r15      //R15 = Salta a l'adreça de retorn
```

FIGURA 73 – Codi CA_assembler del programa Cerca Màxim d'un Vector.

Les simulacions següents mostren part de l'execució del programa cerca el màxim d'un vector de 10 enters. En la primera simulació es poden veure aquelles senyals que encaminen les dades del bloc Program Counter amb l'objectiu de realitzar salts dins el codi. Les següents simulacions mostren casos extrems d'execució.

El valor de les senyals i busos del CAL16 que es mostren se'ls fa referència amb la següent nomenclatura:

clk : Senyal de rellotge que marca la velocitat del processador.

reset : Senyal de reinici d'execució.

Addr_inst [15:0] : Adreça en decimal referent a la instrucció a executar, valor del registre PC.

Ram [11,15:0] : Posició de la Memòria de Dades on s'emmagatzema el màxim al final del programa.

out_R10 [15:0] : Valor en hexadecimal del registre R10 que s'utilitza com a registre temporal durant tot el programa per emmagatzemar el màxim trobat fins el moment.

out_R15 [15:0] : Valor en decimal del registre R15 que s'utilitza per a realitzar les crides a la funció auxiliar, agafant l'adreça inicial de la funció i després emmagatzemant l'adreça de retorn a la funció principal.

salta : Senyal generada en el bloc Program Counter que avalua segons l'estat actual del computador activant-se en el cas d'haver de realitzar un salt o amb el valor 0 si per contrari ha de continuar executant el seqüenciament implícit (PC=PC+1).

jmp : Senyal de control del processador generada per la Unitat de Control que s'activa quan la instrucció que s'està executant és una instrucció de salt incondicional.

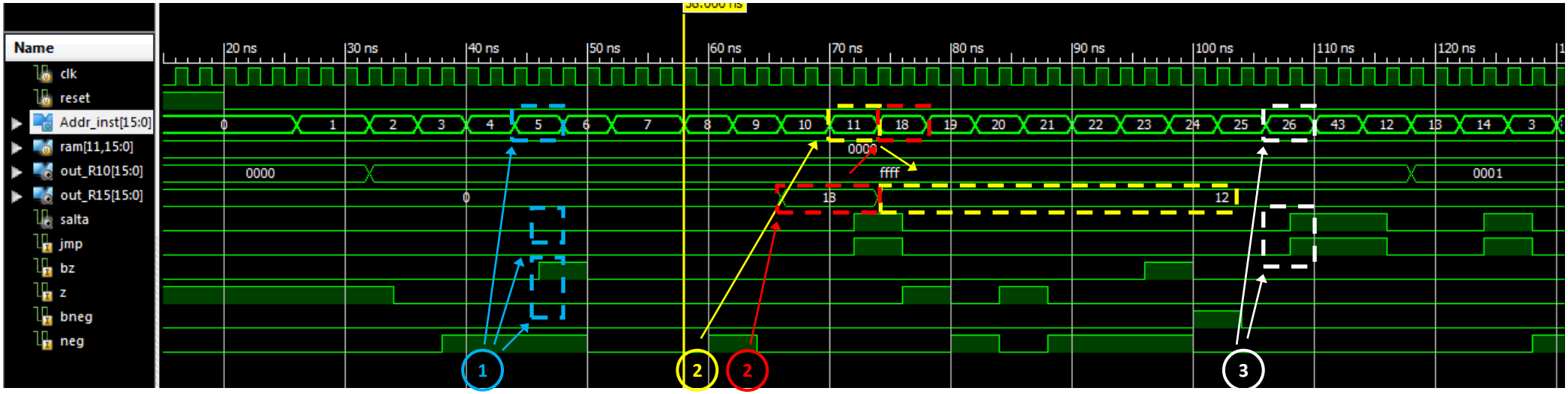
bz : Senyal de control del processador generada per la Unitat de Control que s'activa quan la instrucció que s'està executant és de salt condicional del tipus BZ.

z : Senyal generada pel bloc ALU d'entrada al bloc PC, que permet identificar si el valor de sortida de la ALU és un 0x0000 gràcies a la seva activació.

bneg : Senyal de control del processador generada per la Unitat de Control que s'activa quan la instrucció que s'està executant és de salt condicional del tipus BNEG.

neg : Senyal generada pel bloc ALU d'entrada al bloc PC, que permet identificar si el valor de sortida de la ALU és negatiu gràcies a la seva activació.

- **Simulació 1 inici d'execució:** Dos nombres positius
  Vector [10] = {0xFFFF,0x0001,0xA359,0xC8FF,0x0BD1,0x1234,0x0000,0x8000,0x9898,0x0A34}



1 – Adreça MI = 5, Instrucció = BZ R6,etiq_fifor

Instrucció de salt condicional on no es compleix la condició de salt ja que tal com indica la senyal z (z=0), el valor del registre R6 que passa per la ALU no és igual a 0, per tant la senyal salta no s'activa i es fa seqüenciament implícit (PC=PC+1) i Addr_inst passa a valer 6.

2/2 – Adreça MI = 11, Instrucció = JR R15,R15,0

Instrucció de salt a la funció auxiliar. En vermell podem veure com R15 conté la direcció 18 que és on comença la funció auxiliar. Amb groc podem veure que s'executa la instrucció 11 (JR) que actualitza PC amb la direcció que conté R15 i linka l'adreça de retorn (la actual +1) emmagatzemant-la en R15 (R15 = 11 + 1 = 12).

3 –Adreça MI = 26, Instrucció = JMPI etiq_fi

Instrucció de salt absolut i incondicional, on es pot veure que la senyal jmp, que identifica aquest tipus d'instruccions, està activa i com a conseqüència la senyal salta també s'activa per actualitzar el valor del Registre PC amb la nova direcció (Addr_inst = 43).

- **Simulació 1 final d'execució:** Dos nombres positius
  Vector [10] = {0xFFFF,0x0001,0xA359,0xC8FF,0x0BD1,**0x1234,**0x0000,0x8000,0x9898,0x0A34}

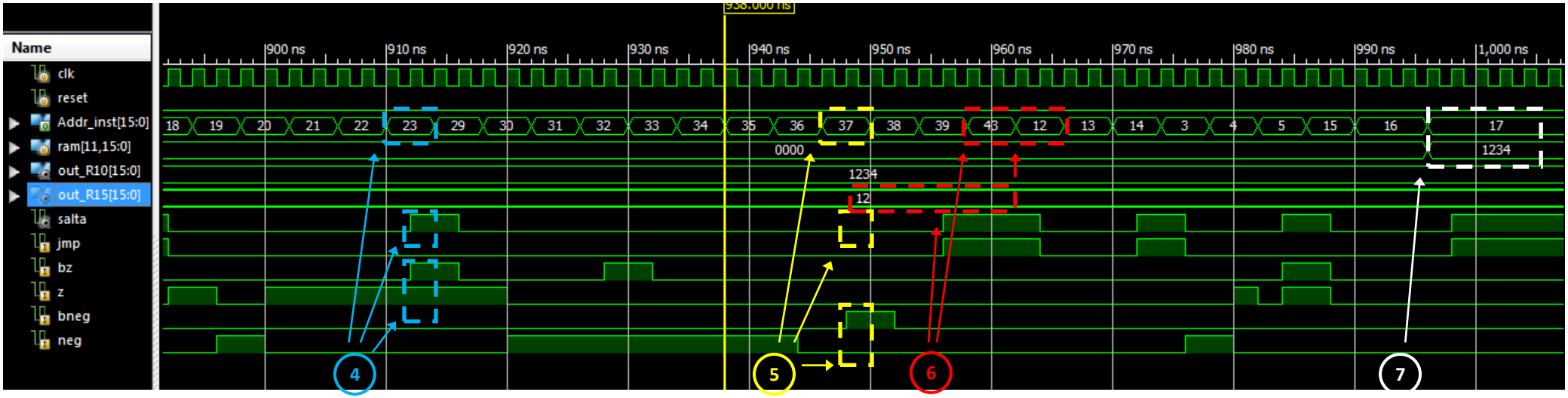


FIGURA 74 – Simulació I del programa Cerca Màxim d'un Vector.

4 – Adreça MI = 23, Instrucció = BZ R3, etiq_else0 — Instrucció de salt relatiu al PC i condicional, en el que es pot veure gràcies a les senyals bz i z que la condició que R3 sigui zero es compleix. Per aquest motiu s'activa la senyal salta que actualitza el registre PC amb el seu valor més l'immediat que correspon (Addr_inst = 29).

5 – Adreça MI = 37, Instrucció = BNEG R3, etiq_else2 — Instrucció de salt condicional on no es compleix la condició de salt ja que tal com indica la senyal neg (neg=0), el valor del registre R3 que passa per la ALU no és negatiu, per tant la senyal salta no s'activa i es fa seqüenciament implícit (PC=PC+1).

6 – Adreça MI = 43, Instrucció = JMP R15 — Instrucció de retorn de la funció màxim, on s'actualitza el valor del PC amb el valor de R15 o hi havia emmagatzemada la direcció de retorn (Addr_inst = 12).

7 – Adreça MI = 17, Instrucció = JMPI etiq_fi — Instrucció de final d'execució on veiem que màxim del vector després de l'execució es troba emmagatzemat en la direcció 11 de la Memòria de Dades i és 0x1234.

- **Simulació 2:** Màxim a la primera posició

  Vector[10]={-1, -2,-2,-2,-4,-3,-2,-2,-2,-3}

■ ■ ■

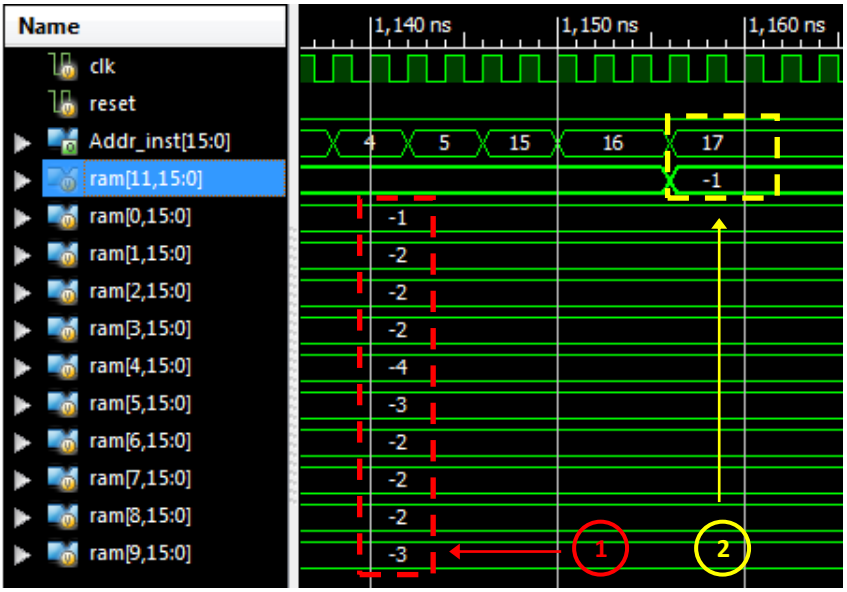


FIGURA 75 – Simulació 2 del programa Cerca Màxim d'un Vector.

1 – Valors inicials vector.

2 – Adreça MI = 17, Instrucció = JMPI etiq_final Final d'execució el resultat MemData[11] = -1.

- **Simulació 3:** Màxim a la última posició

  Vector[10]={0,0,0,6,0,9,2,3,4,32}

■ ■ ■

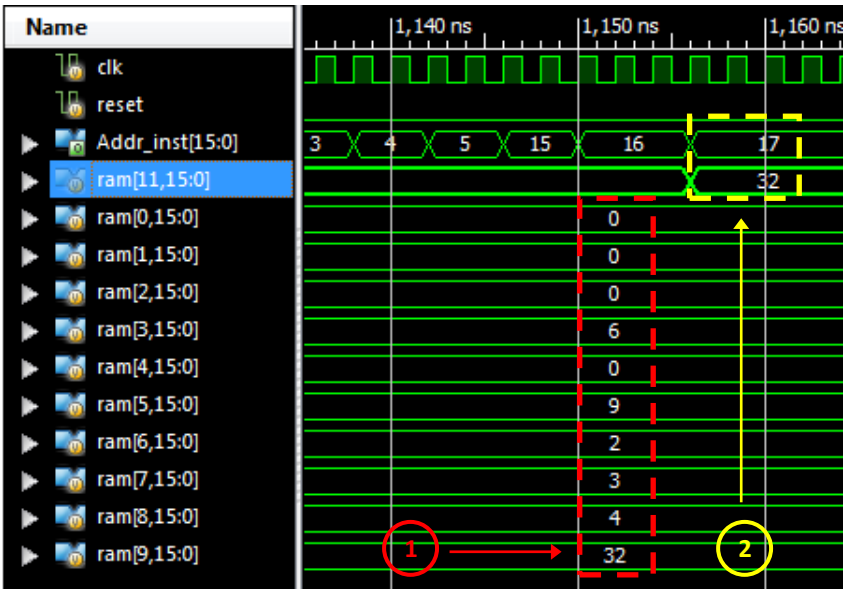


FIGURA 76 – Simulació 3 del programa Cerca Màxim d'un Vector.

1 – Valors inicials vector.

2 – Adreça MI = 17, Instrucció = JMPI etiq_final Final d'execució el resultat MemData[11] = 32.

Exemple 3: Les Torres de Hanoi

ENUNCIAT: Cal moure la pila de peces del pal que es troben a una pal diferent. Només es pot moure un disc cada vegada, i no pot haver-hi mai un disc més gran posat sobre d'un de més petit. Suposem que numerem els pals d'esquerra a dreta (1, 2 i 3). Fer un programa que realitzi els moviments s'han de fer per moure la pila de discos des del pal 1 al pal 3. El nombre de peces és 3 i els seus identificadors són 1, 2, 3, aquests identifiquem també el pes de cada peça, així la peça 3 mai pot estar per damunt de la peça 2.

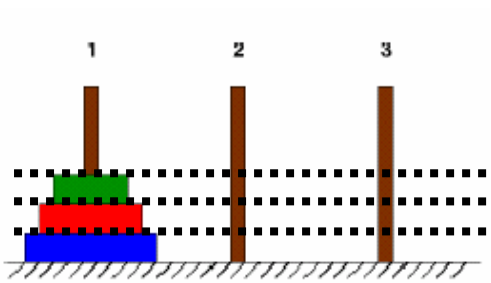
OBJECTIU: Amb aquest exemple es vol mostrar la implementació d'una crida recursiva i mostrar un exemple de com utilitzar la memòria de dades com a pila per emmagatzemar dades temporals.

* Per implementar la funció recursiva cal en cada execució d'aquesta, emmagatzemar a la pila la pròpia direcció de retorn, ja que sempre es linka sobre R15 i es sobreescriuria aquest valor. Els paràmetres es passen pels registres R1, R2, R3 i R4 respectivament.

* Per la utilització de la pila, en aquest exemple s'ha definit un registre com a punter al top de la d'aquesta. En el programa Les Torres de Hanoi s'utilitza les últimes posicions de la Memòria de Dades com espai de la pila i el registre R14 com a punter. R14 s'inicialitza amb el valor 256 (significant que la pila està buida, el punter apunta fora de la memòria) i en cada inserció a la pila primer s'actualitza el punter per aconseguir una posició nova de memòria i després s'emmagatzema el valor temporal. La lectura correspon als passos contraris, primer es llegeix el top i llavors s'actualitza el punter que passa a apuntar a la següent dada no llegida.

REPRESENTACIÓ: Per representar Les Torres de Hanoi, s'utilitza una matriu a les primeres posicions de la Memòria de Dades, on aquesta té 3 columnes, tantes com pals disposa l'exercici, i tantes files com nombre de peces a moure hi ha.

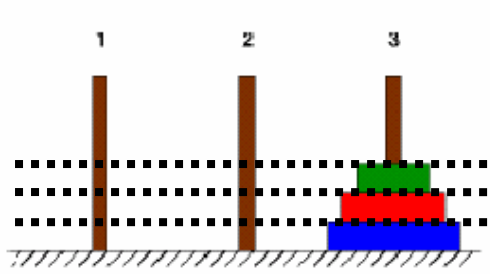
En aquest exemple es realitza l'execució de Les Torres de Hanoi de 3 pals amb 3 peces a moure des del pal1 al pal3, utilitzant el pal2 com a auxiliar. Les peces estan numerades de 1 a 3, on aquest nombre simbolitza el seu pes. El valor 0 a una posició de memòria significa que no hi ha cap peça. El següent dibuix esquematitza la representació:



1	0	0
2	0	0
3	0	0

MemData[0]	3
MemData[1]	2
MemData[2]	1
MemData[3]	0
MemData[4]	0
MemData[5]	0
MemData[6]	0
MemData[7]	0
MemData[8]	0

FIGURA 77 – Exemple inici execució programa Les Torres de Hanoi.



0	0	1
0	0	2
0	0	3

MemData[0]	0
MemData[1]	0
MemData[2]	0
MemData[3]	0
MemData[4]	0
MemData[5]	0
MemData[6]	3
MemData[7]	2
MemData[8]	1

FIGURA 78 – Exemple final execució programa Les Torres de Hanoi.

{ Pre: Les 9 primeres posicions de Memòria de Dades representen la Torra de Hanoi amb 3 peces. Les peces es troben en el pal 1 ordenades segons el seu pes. En la resta de pals no hi ha cap peça. }

{ Post: Les peces es troben en el pal 3 ordenades segons el seu pes, i els moviments realitzats no han violat les restriccions de moviments definides per l'enunciat. }

```
void hanoi(int numPeces, char* R2, char* R3, char* R4) {
    if (numPeces == 1) *R4 = *R2;
    else {
        hanoi(numPeces - 1, R2, R4, R3);
        *R4 = *R2;
        hanoi(numPeces - 1, R3, R2, R4);
    }
}
void main(int discos) {
    int numPeces=3;
    p1 = &MemData[2];   //Pal ple apunta al top
    p2 = &MemData[2];   //Pal buit apunta sota la primera pos
    p3 = &MemData[2];   //Pal buit apunta sota la primera pos
    hanoi(numPeces, p1, p2, p3);
}
```

FIGURA 79 – Codi en C del programa Les Torres de Hanoi.

```
//R1 = nombre de roscos = 3
//R2 = punter al top del PAL1  -> PAL1: Mem[0,1,2]
//R3 = punter al top del PAL2  -> PAL2: Mem[3,4,5]
//R4 = punter al top del PAL3  -> PAL3: Mem[6,7,8]

etiq_main:
 0:  LI R14,1
 1:  ROTR R14,R14,8    //R14 = &pila
 2:  LI R1,3           //R1 = 3, nombre de peces
 3:  LI R2,2           //R2 = &top pal 1  (ple)
 4:  LI R3,2           //R3 = &inicial  pal 2  (buit)
 5:  LI R4,5           //R2 = &inicial  pal 3  (buit)
 6:  LI R15, etiq_hanoi
 7:  LIh R15, etiq_hanoi
 8:  JR R15,R15,0     //Salt a la funció recursiva
etiq_final:
 9:  JMPI etiq_final

etiq_hanoi:
 10: ADDI R1,R1,-1
 11: ADDI R14,R14,-1
 12: ST 0(R14),R15     //Guardar & retorn a la pila
 13: ADDI R5,R4,0
 14: ADDI R4,R3,0      // Pas de paràmetres
 15: ADDI R3,R5,0      //Pas de paràmetres
 16: LI R15, etiq_hanoi
 17: LIH R15, etiq_hanoi  //Salt a la funció recursiva
 18: JR R15,R15,0      //Salt a la funció recursiva
 19: ADDI R5,R4,0
 20: ADDI R4,R3,0      //Recol·locar  paràmetres
 21: ADDI R3,R5,0      // Recol·locar  paràmetres
 22: LD R5,0(R2)       //Treure peça
 23: LI R6,0
 24: ST 0(R2),R6       //Posició buida
 25: ADDI R2,R2,-1     //Actualitzar punter
 26: ADDI R4,R4,1      //Actualitzar punter
 27: ST 0(R4),R5       //Canviar peça de pal
 28: ADDI R5,R3,0
 29: ADDI R3,R2,0      // Pas de paràmetres
 30: ADDI R2,R5,0      // Pas de paràmetres
 31: LI R15, etiq_hanoi
 32: LIH R15, etiq_hanoi
 33: JR R15,R15,0      //Salt a la funció recursiva
 34: ADDI R5,R3,0
 35: ADDI R3,R2,0      //Recol·locar  paràmetres
 37: ADDI R2,R5,0      //Recol·locar  paràmetres
 38: LD R15,0(R14)     //Restaurar &retorn
 39: ADDI R14,R14,1
 40: ADDI R1,R1,1      //Restaurar nombre peces
 41: JMPI etiq_fiif
etiq_casBase:
 43: LD R5,0(R2)       //Treure peça
 44: LI R6,0
 45: ST 0(R2),R6       //Posició buida
 46: ADDI R2,R2,-1     //Actualitzar punter
 47: ADDI R4,R4,1      //Actualitzar punter
 48: ST 0(R4),R5       //Canviar peça de pal
etiq_fiif:
 49: JMP R15           //Salta a l'& de retorn
```

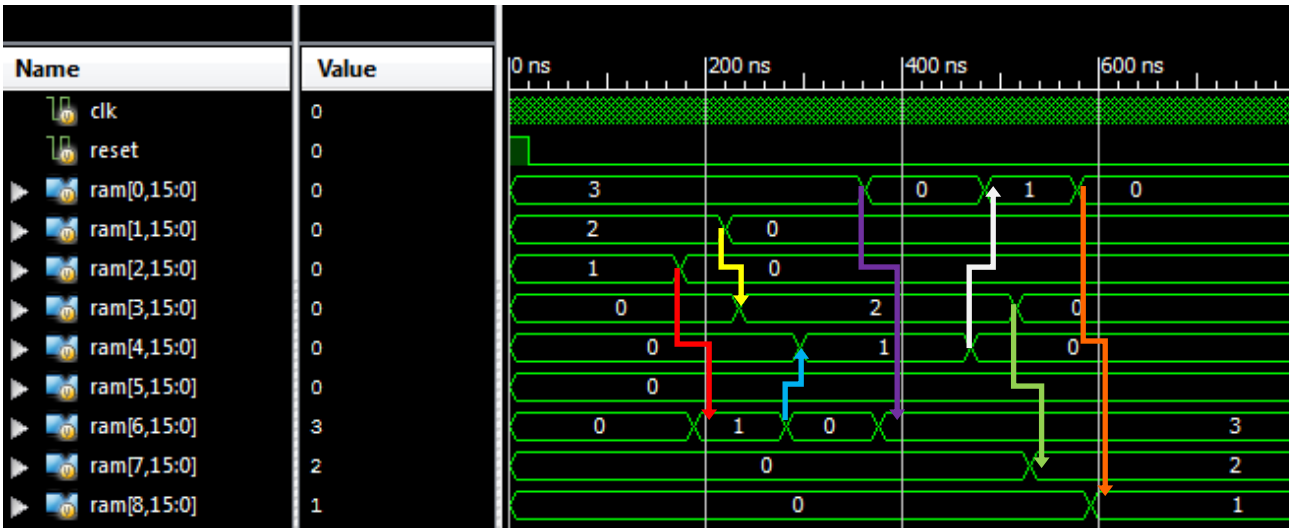
FIGURA 80 – Codi en CAL_assembler del programa Les Torres de Hanoi.

La següent simulació mostra part de l'execució del programa Les Torres de Hanoi. El valor de les senyals i busos del *CAL16* que es mostren se'ls fa referència amb la següent nomenclatura:

clk : Senyal de rellotge que marca la velocitat del processador.

reset : Senyal de reinici d'execució.

Ram [x,15:0] : Valor en decimal de les posicions de Memòria de Dades (on x = [0 .. 8]) que representen l'estructura de Les Torres de Hanoi definida en Representació.



Moviments realitzats:

→	Origen: Pal 1	Destí: Pal 3	Objecte: Peça 1
→	Origen: Pal 1	Destí: Pal 2	Objecte: Peça 2
→	Origen: Pal 3	Destí: Pal 2	Objecte: Peça 1
→	Origen: Pal 1	Destí: Pal 3	Objecte: Peça 3
→	Origen: Pal 2	Destí: Pal 1	Objecte: Peça 1
→	Origen: Pal 2	Destí: Pal 3	Objecte: Peça 2
→	Origen: Pal 1	Destí: Pal 3	Objecte: Peça 1

FIGURA 81 – Simulació del programa Les Torres de Hanoi.

SPARTAN 3E

El projecte fa ús d'una Field Programmable Gate Array (FPGA) per a simular el comportament del processador *CAL16*. Més concretament s'ha fet ús del paquet Spartan-3E High Volume Starter Kit, que dona accés a la plataforma completa de la família Spartan 3E, que és la que ve determinada segons la FPGA que s'utlitza.

El conjunt d'eines que formen aquest entorn, és una solució completa per la construcció de projectes utilitzant les funcions de la familia Spartan 3E. Aquest inclou:

- Espai de desenvolupament.
- Font d'alimentació universal 100-240V, 50/60Hz.
- ISE WebPack software i Embedded Development Kit (EDK).
- Manual d'introducció al disseny de la lògica programable.
- CD de recursos (Starter Kit).
- Cable USB.

FIGURA 82 – FPGA Spartan 3E

Com a característiques principals, tot i que no han estat totes utilitzades en el projecte però hi ha la possibilitat d'incorporar-les en properes ampliacions, la FPGA Spartan 3E disposa de:

Dispositius de Xilinx

- Spartan-3E FPGA (XC3s500E-4FG320C)
- CPLD CoolRunner ™-II (XC2C64A-5VQ44C) Plataforma Flash (XCF04S-VO20C)

Rellotge:

- 50 MHz

Memòria:

- 128 Mbit Flash en paral·lel.
- 16 Mbit Flash SPI 64 Mbytes de SDRAM DDR .

Connectors i interfícies:

- Ethernet 10/100 PHY .
- JTAG USB descàrrega.
- Dos Ports Sèrie de 9 pins RS-232 .
- PS/2- estil port de ratolí o teclat.
- Codificador rotatori amb polsador.
- 4 interruptors de desplaçament.
- 8 sortides led individuals.
- 4 botons de contacte momentani.
- Ports d'expansió de 100 pins .
- 3 connectors d'expansió de 3 pins.

Pantalla

- LCD de 2 línies de 16 caràcters.

EXECUCIÓ

Els programes presentats com a proves en aquest apartat, estan executats en la FPGA, per aquest motiu, l'única via de comprovació dels resultats és a través de l'espai de sortida del *CAL16*, que està directament connectat al dispositiu dels leds de la placa.

Per a cada exemple s'utilitza el mateix format de presentació que en l'apartat anterior, amb una descripció de l'objecte d'estudi, la presentació del programa a executar i per a mostrar els resultats s'adjunta un CD on es poden trobar els vídeos corresponents a l'execució de cadascun[1].

Exemple 4: Onada de leds

OBJECTIU: Amb aquest exemple es vol mostrar l'accés a l'espai de sortida del processador directament connectat als leds de la placa.

Per implementar-ho es fa ús d'una funció encarregada de deixar passar 1 segon d'espera amb l'objectiu de permetre veure els canvis en els leds, en cas de no esperar-se aquest segon l'execució seria molt ràpida i no permetria apreciar els canvis.

{ Pre: -. }

{ Post: Els leds s'activen de forma seqüencial per ordre de pes, de manera que tant sols un led està encès cada segon i el següent a encendre's és el de la seva esquerra fins arribar al de més pes, on es torna a començar pel de menys pes. }

```
void esperaUs(){
    for(i=0;i<(2^10);i++) {
        for(j=0;j<(2^11);j++){}
    }
}

void onadaDeLeds(){
    leds = 1;
    while (true){
        leds = leds rotr 1;
        if(leds == 256) leds = 1;
        esperaUs();
    }
}
```

FIGURA 83 – Codi en C del programa Onada de Leds.

[1] *CD Adjunt: Dins el CD adjunt, en el directori Demos\ hi ha el conjunt de vídeos de demostració. Per més informació veure capítol Annexes, Annex 3.*

```
0:  LI R1,1
1:  LI R0,0
2:  LIH R0,1
3:  ST 0(R0),R1          //leds = 1
etiq_while:
4:  LI R3,etiq_esperals
5:  LIH R3,etiq_esperals
6:  JR R15,R3,0          //espera 1 segon
7:  LD R2,0(R0)
8:  ROTR R2,R2,1         //R2 = leds << 1
9:  LI R3,0
10: LIH R3,-1
11: ADD R3,R2,R3
12: BNEG R3, etiq_fiif
13: LI R2,1              //R2 = 1
etiq_fiif:
14: ST 0(R0),R2          //leds = R2
15: JMPI etiq_while
etiq_esperals:
16: LI R10,0             //R10=i=0
17: LI R11,0
18: LIH R11,-4           //R11=0xFC00=-1024
etiq_for1:
19: ADD R9,R10,R11
20: BZ R9, etiq_fifor1
21: LI R12,0             //R12=j=0
22: LI R13,0
23: LIH R13,-8           //R13=0xF800=-2048
etiq_for2:
24: ADD R9,R12,R13
25: BZ R9, etiq_fifor2
26: ADDI R12,R12,1       // j++
27: JMPI etiq_for2
etiq_fifor2:
28: ADDI R10,R10,1       // i++
29: JMPI etiq_for1
etiq_fifor1:
30: JMP R15
```

FIGURA 84 – Codi en CAL_assembler del programa Onada de Leds.

EXEMPLE: El següent cronograma mostra un exemple de funcionament per l'exercici.

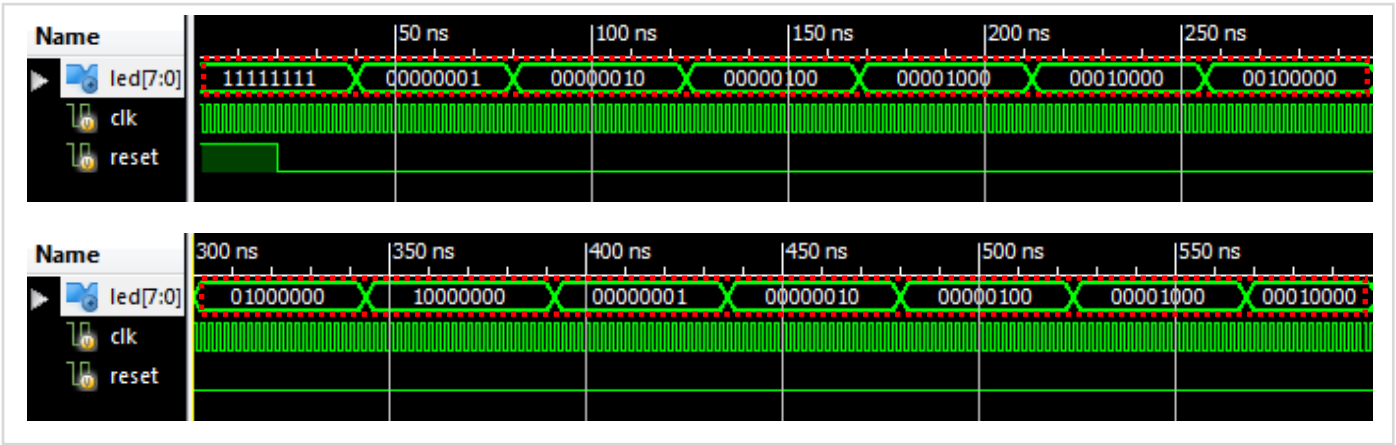


FIGURA 85 – Execució del programa Onada de Leds.

EXECUCIÓ: Per a veure el vídeo que mostra la seva execució amb la FPGA Spartan 3E, veure capítol Annexes, Annex 3. Dins el CD en el directori de Demos, el vídeo s'anomena *OnadaDeLeds.avi.*

Exemple 5: Leds en funció de l'interruptor escollit

OBJECTIU: Amb aquest exemple es vol mostrar la lectura de l'espai d'entrada del *CAL16* el qual està directament connectat als interruptors de la placa.

Per implementar-ho es fa ús d'una funció encarregada de deixar passar 1 segon d'espera amb l'objectiu de permetre llegir els canvis del interruptor sense interferències entremitges, ja que dins el flancs associats al moviment d'un interruptor, entre el valor absolut de 0 i el valor de 1, genera un soroll que es pot mal interpretar com múltiples canvis entremitjos.

{ Pre: La posició MemData[0] equival a la variable ant i està inicialitzada a 0. }

{ Post: Els leds s'activen mostrant el valor que codifiquen els 3 interruptors de menys pes de la placa a cada moviment del interruptor de més pes. }

```
int ant = 0;
void mostraValorSwitch(){
    leds = 0;
    do{
        do{
            act = switches;
        }while (act[3] == ant);
        ant = act[3];
        leds = act[2:0];
        espera1s();
    } while (true);

}

void espera1s(){
    for(i=0;i<(2^10);i++) {
        for(j=0;j<(2^11);j++){}
    }
}
```

FIGURA 86 – Codi en C del programa Leds en funció del Switch.

```
//MemData[0x00] = ant = 0
 0:  LI R1,0          //R1 = &ant
 1:  LI R0,0
 2:  LIH R0,1         //R0 = &espai e/s
 3:  ST 0(R0),R1      //leds = 0
 4:  ST 0(R1),R1      //ant = 0
etiq_do:
 5:  LD R4,0(R1)
etiq_enquesta:
 6:  LD R2,1(R0)      //R2 = act = switches
 7:  LI R3,8
 8:  AND R3,R3,R2     //R3 = act[3]
 9:  ROTR R3,R3,13
10:  XOR R5,R3,R4
11:  BZ R5,etiq_enquesta
12:  ST 0(R1),R3      //ant=act[3]
13:  LI R5,8
14:  LI R6,-1
15:  XOR R5,R5,R6
16:  AND R5,R5,R2     //act[2:0]
17:  ST 0(R0),R5
18:  LI R3,etiq_esperals
19:  LIH R3,etiq_esperals
20:  JR R15,R3,0      //espera 1 segon
21:  JMPI etiq_do
etiq_esperals:
22:  LI R10,0         //R10=i=0
23:  LI R11,0
24:  LIH R11,-4       //R11=0xFC00=-1024
etiq_for1:
25:  ADD R9,R10,R11
26:  BZ R9, etiq_fifor1
27:  LI R12,0         //R12=j=0
28:  LI R13,0
29:  LIH R13,-8       //R13=0xF800=-2048
etiq_for2:
30:  ADD R9,R12,R13
31:  BZ R9, etiq_fifor2
32:  ADDI R12,R12,1   // j++
33:  JMPI etiq_for2
etiq_fifor2:
34:  ADDI R10,R10,1   // i++
35:  JMPI etiq_for1
etiq_fifor1:
36:  JMP R15
```

FIGURA 87 – Codi en CAL_assembler del programa Leds en funció del Switch.

EXEMPLE: El següent cronograma mostra un exemple de funcionament per l'exercici. En cada canvi del interruptor de més pes mostra el valor pels leds dels interruptors de menys pes.

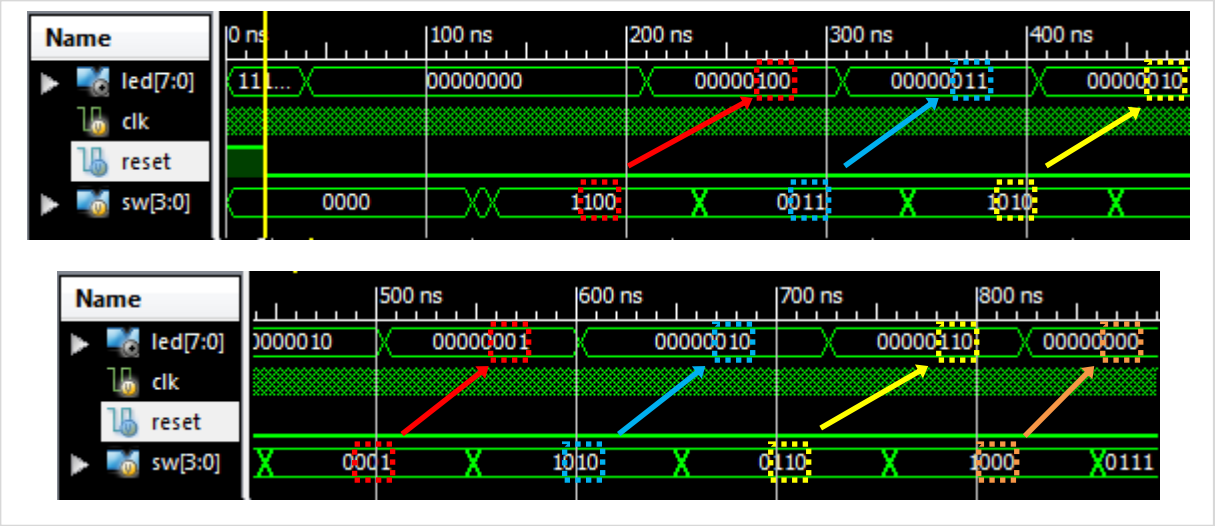


FIGURA 88 – Simulació del programa Leds en funció del Switch.

EXECUCIÓ: Per a veure el vídeo que mostra la seva execució amb la FPGA Spartan 3E, veure capítol Annexes, Annex 3. Dins el CD en el directori de Demos, el vídeo s'anomena *LedsFunciói Switch.avi.*

Exemple 6: Suma N

OBJECTIU: Amb aquest exemple es vol mostrar la interacció entre els espais d'entrada i sortida del *CAL16*.

Per implementar-ho es fa ús d'una funció encarregada de deixar passar 1 segon d'espera amb l'objectiu de permetre llegir els canvis del interruptor sense interferències entremitges, ja que dins el flancs associats al moviment d'un interruptor, entre el valor absolut de 0 i el valor de 1, genera un soroll que es pot mal interpretar com múltiples canvis entremitjos.

{ Pre: El primer nombre a llegir és N, el número de nombres que es processaran. N > 0. }

{ Post: Els leds mostren el valor codificat en binari resultant de sumar els N nombres llegits per l'entrada, on l'entrada d'un nou nombre s'identifica pel canvi de flanc del interruptor de més pes i el nombre es troba codificat mitjançant els 3 interruptors de menys pes. }

```
int ant = 1;
void sumaN(){
    while(!switches[3]){}
    N = switches[2:0];
    suma = 0;
    for(int i=N;i>0;i--){
        espera1s();
        while(switches[3] == ant){}
        ant = switches[3];
        suma = suma + switches[2:0];
    }
    leds = suma;
}

void espera1s(){
    for(i=0;i<(2^10);i++) {
        for(j=0;j<(2^11);j++){}
    }
}
```

FIGURA 89 – Codi en C del programa Suma N.

```
//MemData[0x00] = ant = 0
 0:  LI R1,0              //R1 = &ant
 1:  LI R2,1
 2:  ST 0(R1),R2          //ant = 1
 3:  LI R0,0
 4:  LIH R0,1             //R0 = &espai e/s
 5:  ST 0(R0),R1          //leds = 0
etiq_enquesta1:
 6:  LD R2,1(R0)          //R2 = act = switches
 7:  BZ R2,etiq_enquesta1
 8:  LI R3,7
 9:  AND R3,R2,R3         //R3=N=switches[2:0]
 10: XOR R4,R4,R4         //R4=suma=0
etiq_for:
 11: BZ R3,etiq_fi        //i>0
 12: LI R5,etiq_esperals
 13: LIH R5,etiq_esperals
 14: JR R15,R5,0          //espera 1 segon
 15: LD R5,0(R1)          //R5 = ant
etiq_enquesta2:
 16: LD R2,1(R0)          //R2 = act = switches
 17: LI R6,8
 18: AND R6,R6,R2         //R6 = act[3]
 19: ROTR R6,R6,13
 20: XOR R7,R6,R5
 21: BZ R7,etiq_enquesta2
 22: ST 0(R1),R6//ant=act[3]
 23: LI R5,7
 24: AND R5,R2,R5         //R5=switches[2:0]
 25: ADD R4,R4,R5         //suma+=nombre
 26: ADDI R3,R3,-1        //i--
 27: JMPI etiq_for
etiq_fi:
 28: ST 0(R0),R4
 29: JMPI etiq_fi

etiq_esperals:
 30: LI R10,0             //R10=i=0
 31: LI R11,0
 32: LIH R11,-4           //R11=0xFC00=-1024
etiq_for1:
 33: ADD R9,R10,R11
 34: BZ R9, etiq_fifor1
 35: LI R12,0             //R12=j=0
 36: LI R13,0
 37: LIH R13,-8           //R13=0xF800=-2048
etiq_for2:
 38: ADD R9,R12,R13
 39: BZ R9, etiq_fifor2
 40: ADDI R12,R12,1       // j++
 41: JMPI etiq_for2
etiq_fifor2:
 42: ADDI R10,R10,1       // i++
 43: JMPI etiq_for1
etiq_fifor1:
 44: JMP R15
```

FIGURA 90 – Codi en CAL_assembler del programa Suma N.

EXEMPLE: El següent cronograma mostra un exemple de funcionament per l'exercici. En aquest cas N = 4, i els nombres que entren per a ser processats són 3,2,1,2. El resultat mostrat pels leds és 3+2+1+2 = 8.

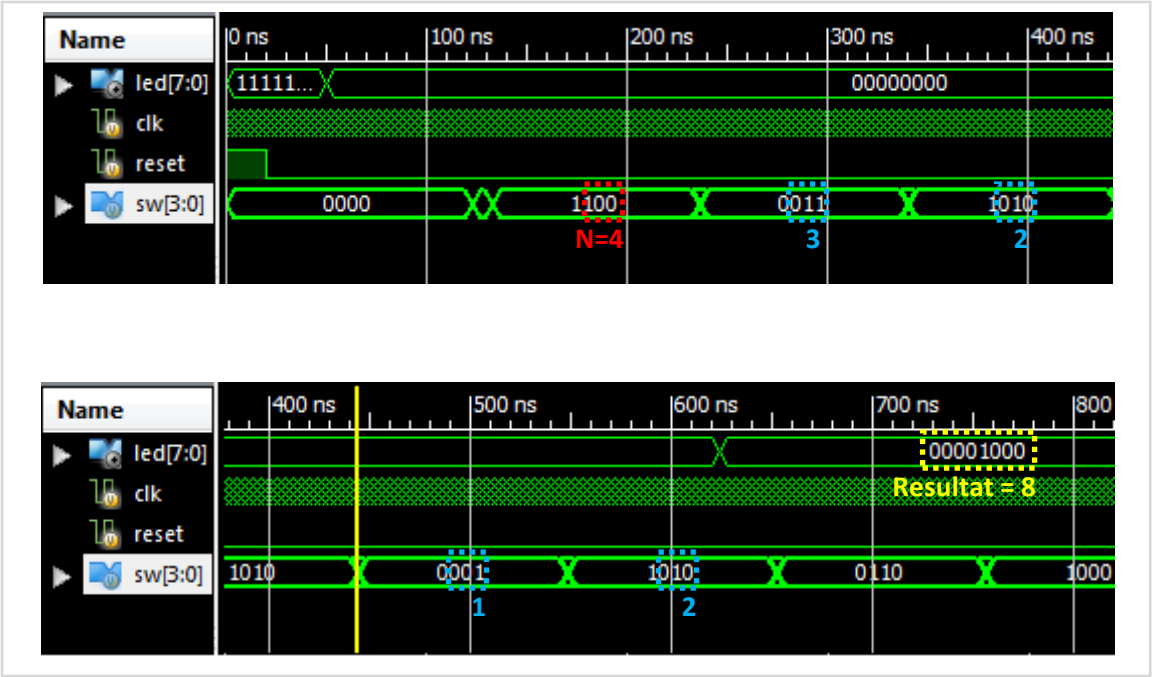


FIGURA 91 – Execució del programa Suma N.

EXECUCIÓ: Per a veure el vídeo que mostra la seva execució amb la FPGA Spartan 3E, veure capítol Annexes, Annex 3. Dins el CD en el directori de Demos, el vídeo s'anomena *SumaN.avi.*

Documents Adjunts

INTRODUCCIÓ AL VERILOG

ÍNDEX

MANUAL D'INTRODUCCIÓ AL VERILOG

INTRODUCCIÓ

Per a la implementació del processador *CAL16* s'utilitza el llenguatge Verilog. Per aquest motiu, aquest apartat de la memòria té com a objectiu introduir aquest llenguatge i algunes de les possibilitats de programació que ofereix.

Aquest apartat no presenta un manual complet de com programar en Verilog, sinó més aviat una simplificació. A continuació es troba tot allò referent al llenguatge Verilog que es necessita, per a comprendre com està implementat el processador *CAL16*, per a permetre la seva modificació o ampliació i per la creació de nous blocs seguint el mateix estil de programació.

DEFINICIÓ

Verilog és un llenguatge de descripció de Hardware (HDL) utilitzat per a modelar sistemes electrònics.

La seva programació es basa en la declaració de busos i senyals interconnectats entre ells mitjançant portes lògiques per a formar mòduls, que alhora es poden connectar amb d'altres. Permet utilitzar una sintaxis similar a la del llenguatge C fent ús de mecanismes de control com; if, while, etc.., tot i que en alguns aspectes, com en la encapsulació de mòduls o definició de constants, se'n diferencia.

L'execució de sentències en Verilog no és estrictament lineal, ja que en un projecte Verilog disposem d'una jerarquia mòduls els quals estan connectats entre si mitjançant ports d'entrada sortida i cables, aquest fet implica que les senyals es propaguen d'un bloc a l'altre sense permetre garantir la linealitat. Encara que també existeixen mecanismes de control per a intentar garantir l'ordre.

Per a poder programar una FPGA amb un projecte implementat en Verilog, és necessari que aquest compleixi la característica de sintetitzabilitat, és a dir, que es pugui convertir en un llistat de nodes que descriguin els components i connectors que han d'implementar-se a nivell de hardware. Un mòdul és sintetitzable si conté sentències sintetitzables i els mòduls que instancia són sintetitzables.

IMPLEMENTACIÓ

- **NOMBRES:** Per a representar un nombre, és necessari indicar el nombre de bits i la base amb els que es representa.

Nombre de bits	‘	Base	Valor

FIGURA 92 – Format codificació de nombres immediats en Verilog.

En cas de no indicar el nombre de bits, per defecte, es considera 32 bits. En cas de no indicar la base es considera decimal. Les possibles bases de representació són:

Marca de base	Base
‘b ‘B	Binari
‘d ‘D	Decimal
‘h ‘H	Hexadecimal
‘o ‘O	Octal

Exemples:

8’b10100101 4’d5 16’habcd 32’o01234567012

FIGURA 93 – Format i Exemples de codificació de nombres immediats en Verilog en diferents bases..

- **VALORS:** Un bit pot tenir 4 valors diferents

  0 - Baix voltatge
  1 - Alt voltatge
  x - Indeterminat
  z - Alta impedància

  Al representar un nombre, si el número de bits determinat amb el primer dígit de la representació d’un nombre i el número de bits que es col·loquen darrere com a valor no corresponen, Verilog no estén el signe, sinó que afegeix zeros, menys en cas que es tracti del valor x o el valor z on estén x i z respectivament.

Exemples:

8’b0	0000 0000	8’b1	0000 0001
8’bx	xxxx xxxx	8’hz1	zzzz 0001
8’b1x	0000 001x	8’bx1	xxxx xxx1
8’b0x	0000 000x	8’bx0	xxxx xxx0
8’hx	xxxx xxxx	8’hz	zzzz zzzz
8’hzx	zzzz xxxx	8’h0z	0000 zzzz

FIGURA 94 – Exemples de generació automàtica de bits que realitza Verilog per defecte..

- **SENTÈNCIES:** Totes les sentències han d’acabar en punt i coma.

- **COMENTARIS:** El Verilog permet dos tipus de comentaris:

Exemples:

```
// Això és un comentari
/* Això és un comentari que permet salts de línea en mig */
```

FIGURA 95 – Tipus de comentaris que permet Verilog.

- **TIPUS DE DADES:** S'utilitzen els següents tipus.

wire	-	Representació d'un cable utilitzat per a connectar components. No permet emmagatzemar informació.
reg	-	Representació d'una variable que manté el valor fins que li arriba una dada nova.
wire/reg [x : y] (vector)	-	Indica el nombre de bits que el formen. Sempre s'agafa el nombre de l'esquerra com a bit de més pes. **Ex: wire[7:0] nom** Per accedir a un bit en concret **nom[3]**
input	-	Port d'entrada, per defecte tractat com a wire.
output	-	Port de sortida, per defecte tractat com a wire tot i que també es pot declarar com a reg.
inout	-	Port d'entrada sortida, per defecte tractat com a wire.

FIGURA 96 – Tipus de dades que permet Verilog.

- **MÒDULS:** Tots els mòduls han d'anar encapsulats seguint el següent esquema:

Exemple:

```
module nom_modul ( param1, param2, param3);
        input param1, param2;
        output param3;
        ...
endmodule
```

FIGURA 97 – Estructura d'un mòdul en Verilog.

- **PROCESSOS:** Existeixen dos tipus de processos en Verilog:

  Initial: S'executa una sola vegada en el moment que el temps és 0, és a dir, al inici.
  Always: Procés que s'executa constantment.

Exemples:

```
module Ram   (  input clk, input [7:0] address, input [15:0] data_in,
                output reg [15:0] data_out, input w, input cs);

    reg [15:0] ram [0:255];

    always @(posedge clk)
        data_out <= ram[address];

    always @(posedge clk)
        if (w && cs) ram[address] <= data_in;

     initial
        $readmemb("Programes/data.bin", ram);

endmodule
```

FIGURA 98 – Exemples de processos que permet Verilog.

- **ESDEVENIMENTS:** Un esdeveniment es produeix amb un canvi de valor d'una variable.

Exemples:

Esdeveniment	Descripció
always @(x) z<= x + y;	Cada cop que el valor del senyal x canvia el procés s'avalua.
always @(x or y or z) t <= x + y + z	Cada cop que el valor del senyal x o y canvia el procés s'avalua.
always @(posedge clk) z<= z + y;	Cada cop que es produeix un flanc ascendent de clk el procés s'avalua.
always @(posedge clk or negedge irt) z<= z+ y;	Cada cop que es produeix un flanc ascendent de clk o descendent de irt el procés s'avalua.

FIGURA 99 – Exemples d'esdeveniments que permet Verilog.

- **OPERACIONS:** A continuació es pot trobar un llistat de les operacions més utilitzades.

Exemples:

Tipus	Símbol	Operació	Tipus	Símbol	Operació
Aritmètiques	*	Multiplicar	Lògiques	||	Or
	/	Dividir		!	Negació
	+	Sumar	Relacionals	>	Més gran que
	-	Restar		<	Menor que
	%	Mòdul	Igualtat	==	Iguals
Bit a bit	&	And bit a bit		!=	Diferents
	|	Or bit a bit	Desplaçaments	>>	A la dreta
	^	Xor bit a bit		<<	A l'esquerra
	~	Not bit a bit	Concatenació	{a,b}	Concatenar
Lògiques	&&	And	Condicional	?:	Condició? <sentencia_cert>: <sentencia_fals>

FIGURA 100 – Llistat d'instruccions més utilitzades que permet Verilog.

PROCEDIMENTS: Els procediments que executin més d'una sentència dins seu, han d'anar encapsulats dins les paraules clau begin i end. Les sentències no tenen perquè executar-se de forma seqüencial.

Exemple:

```
begin
        [sentències]
end
```

FIGURA 101 – Exemple de l'estructura d'un conjunt de sentències dins d'un procediment en Verilog.

- **ASSIGNACIONS:** Per assignar un valor a una variable, l'assignació depèn del tipus de dades al que estem assignant.

Exemples:

Assignació	Descripció

initial begin
i = 2;
j = 3;
#1 i = 4;
k = #1 j;
#3 i = #3 j;
#3 j = #3 i;
k<= #5 j;
end
	Assignacions dins un procediment:
	La primera i la segona assignació són simples, i s'executa en temps = 0.
	La tercera, espera a temps = 1 per realitzar l'avaluació del valor a assignar i l'assignació.
	A la quarta l'avaluació és immediat, però l'assignació es retarda fins a temps = 1.
	A la cinquena i sisena, tant i com j són avaluades en temps = 1 i assignades en temps = 2.
	L'última assignació es tracta d'una assignació no bloquejant, és a dir, s'assigna a k en temps = 5, però el procés ha acabat en temps = 3, ja que l'última assignació no el bloqueja.

assign data_act = leds[7:0];	**Connexions:** S'utilitza en tipus de dades wire, es pot veure com l'acció de connectar el cable. A partir d'ara data_act sempre transportarà el mateix valor que tingui emmagatzemat leds.

FIGURA 102 – Exemple de diferents assignacions possibles en Verilog.

- **INSTÀNCIES:** A continuació es mostra un exemple de com instanciar un mòdul dins un altre. **nom_modul *nom_instancia* (connexions d'entrada i de sortida);**

Exemple:

```
module mux2_1_1bit(a, b, sel, out);
        output out;
        input a,b,sel;
        not I5 (sel_n, sel);
        and I6 (sel_a, a, sel);
        and I7 (sel_b, b, sel_n);
        or I4 (out, sel_a, sel_b);
endmodule

module mux2_1_4bit(a, b, sel, out);
        output [3:0] out;
        input [3:0] a,b;
        input sel;
        mux2_1_1bit bit0 (a[0], b[0], sel, out[0]);
        mux2_1_1bit bit1 (a[1], b[1], sel, out[1]);
        mux2_1_1bit bit2 (a[2], b[2], sel, out[2]);
        mux2_1_1bit bit3 (a[3], b[3], sel, out[3]);
endmodule
```

FIGURA 103 – Exemple de com instanciar diferents blocs en Verilog.

- **MECANISMES DE CONTROL DE FLUX:** Verilog permet l'ús de mecanismes de control de flux, sempre dins d'un procediment, com els utilitzats en C, tals com:

```
if (condició) begin
    sentència;
    sentència;
        ....
end

if (condició) sentència;
else sentència;

if (condició)
    if (condició) sentència;
else sentència;

case (sel)
   2'b00: sentència;
   2'b01: sentència;
   2'bx0: sentència;
   default: sentència;
```

```
if (condició)  sentència;

forever sentència;

while (condició) begin
    sentència;
    sentència;
        ....
end

for ( assignació; condició; assignació) begin
    sentència;
    sentència;
        ....
end
```

FIGURA 104 – Exemples de mecanismes de control de flux que permet Verilog.

*En tots els exemples anteriors en cas d'haver més d'una sentència dins el mecanisme de control, caldrà encapsular-les dins les paraules **begin [sentencies] end**.

- **TASQUES DE SISTEMA:** Existeixen diferents tasques de sistema que s'identifiquen pel **$** que precedeix el seu nom. En aquest cas, només es fa referència **$readmemb.**

  Aquesta funció s'encarrega de inicialitzar la memòria indicada amb el contingut binari que conté un fitxer .bin

  $readmemb ("ruta_fitxer_binari", instancia_memòria);

Exemple:

```
initial $readmemb ("Programes/program.bin", rom);
```

FIGURA 105 – Exemple d'ús de la tasca $readmemb de Verilog.

EXEMPLES

MÒDULS COMBINACIONALS

A continuació es mostra la implementació d'alguns dels mòduls combinacionals programats en Verilog que utilitza el processador CAL16. Es comença per la implementació de mòduls d'un sòl bit, i es mostra algun exemple de com programar mòduls més grans a partir d'aquests.

AND_1b

```
module and_1b(a,b,w);
   input a,b;
   output w;
   assign w = a&b;
endmodule
```

x	y	w
0	0	0
0	1	0
1	0	0
1	1	1

FIGURA 106 – Implementació d'una porta lògica AND en Verilog.

OR_1b

```
module or_1b(a,b,w);
   input a,b;
   output w;
   assign w = a|b;
endmodule
```

x	y	w
0	0	0
0	1	1
1	0	1
1	1	1

FIGURA 107 – Implementació d'una porta lògica OR en Verilog.

XOR_1b

```
module xor_1b(a,b,w);
   input a,b;
   output w;
   assign w = a^b;
endmodule
```

x	y	w
0	0	0
0	1	1
1	0	1
1	1	0

FIGURA 108 – Implementació d'una porta lògica XOR en Verilog.

NOT_1b

```
module not_1bit (a,c);
  input a;
  output c;
  assign c = ~a;
endmodule
```

x	w
0	1
1	0

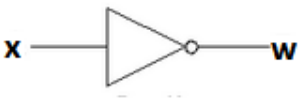


FIGURA 109 – Implementació d'una porta lògica NOT en Verilog.

AND_16b

```
module AND_16b(A,B,W);
   input [15:0] A,B;
   output [15:0] W;

   and_1b bit0 (A[0],B[0],W[0]);
   and_1b bit1 (A[1],B[1],W[1]);
   and_1b bit2 (A[2],B[2],W[2]);
   and_1b bit3 (A[3],B[3],W[3]);
   and_1b bit4 (A[4],B[4],W[4]);
   and_1b bit5 (A[5],B[5],W[5]);
   and_1b bit6 (A[6],B[6],W[6]);
   and_1b bit7 (A[7],B[7],W[7]);
   and_1b bit8 (A[8],B[8],W[8]);
   and_1b bit9 (A[9],B[9],W[9]);
   and_1b bit10 (A[10],B[10],W[10]);
   and_1b bit11 (A[11],B[11],W[11]);
   and_1b bit12 (A[12],B[12],W[12]);
   and_1b bit13 (A[13],B[13],W[13]);
   and_1b bit14 (A[14],B[14],W[14]);
   and_1b bit15 (A[15],B[15],W[15]);
endmodule
```

FIGURA 110 – Implementació del bloc AND de 16 bits en Verilog.

OR_16b

```
module OR_16b(A,B,W);
   input [15:0] A,B;
   output [15:0] W;

   or_1b bit0 (A[0],B[0],W[0]);
   or_1b bit1 (A[1],B[1],W[1]);
   or_1b bit2 (A[2],B[2],W[2]);
   or_1b bit3 (A[3],B[3],W[3]);
   or_1b bit4 (A[4],B[4],W[4]);
   or_1b bit5 (A[5],B[5],W[5]);
   or_1b bit6 (A[6],B[6],W[6]);
   or_1b bit7 (A[7],B[7],W[7]);
   or_1b bit8 (A[8],B[8],W[8]);
   or_1b bit9 (A[9],B[9],W[9]);
   or_1b bit10 (A[10],B[10],W[10]);
   or_1b bit11 (A[11],B[11],W[11]);
   or_1b bit12 (A[12],B[12],W[12]);
   or_1b bit13 (A[13],B[13],W[13]);
   or_1b bit14 (A[14],B[14],W[14]);
   or_1b bit15 (A[15],B[15],W[15]);
endmodule
```

FIGURA 111 – Implementació del bloc OR de 16 bits en Verilog.

XOR_16b

```
module XOR_16b(A,B,W);
   input [15:0] A,B;
   output [15:0] W;

   xor_1b bit0 (A[0],B[0],W[0]);
   xor_1b bit1 (A[1],B[1],W[1]);
   xor_1b bit2 (A[2],B[2],W[2]);
   xor_1b bit3 (A[3],B[3],W[3]);
   xor_1b bit4 (A[4],B[4],W[4]);
   xor_1b bit5 (A[5],B[5],W[5]);
   xor_1b bit6 (A[6],B[6],W[6]);
   xor_1b bit7 (A[7],B[7],W[7]);
   xor_1b bit8 (A[8],B[8],W[8]);
   xor_1b bit9 (A[9],B[9],W[9]);
   xor_1b bit10 (A[10],B[10],W[10]);
   xor_1b bit11 (A[11],B[11],W[11]);
   xor_1b bit12 (A[12],B[12],W[12]);
   xor_1b bit13 (A[13],B[13],W[13]);
   xor_1b bit14 (A[14],B[14],W[14]);
   xor_1b bit15 (A[15],B[15],W[15]);
endmodule
```

FIGURA 112 – Implementació del bloc XOR de 16 bits en Verilog.

DETECTOR_ZERO

```
module DetectorZero_16bit (x,w);
  input [15:0] x;
  output w;
  wire out_n;

  not n1 (out_n, (x[0]|x[1]|x[2]|x[3]|
                  x[4]|x[5]|x[6]|x[7]|
                  x[8]|x[9]|x[10]|x[11]|
                  x[12]|x[13]|x[14]|x[15]));
  assign w = out_n;

endmodule
```

FIGURA 113 – Implementació del bloc Detector de Zero de 16 bits en Verilog.

EXTENSOR_SIGNE

```
module SignExtender_4(in,out);
    input [3:0] in;
    output [15:0] out;

    assign out = {in[3],in[3],in[3],in[3],in[3],in[3],in[3],in[3],in[3],in[3],in[3],in[3],in};

endmodule

module SignExtender_8(in,out);
    input [7:0] in;
    output [15:0] out;

    assign out = {in[7],in[7],in[7],in[7],in[7],in[7],in[7],in};

endmodule

module SignExtender_12(in,out);
    input [11:0] in;
    output [15:0] out;

    assign out = {in[11],in[11],in[11],in[11],in};

endmodule
```

FIGURA 114 – Implementació dels blocs Extensors de Signe cap a 16 bits en Verilog.

DESPLAÇADORS

```
module ShiftRight_16bit(A,r,W);
    input [15:0] A;
    input [3:0] r;
    output [15:0] W;

    assign W = A >> r;
endmodule

module ShiftLeft_16bit(A,r,W);
    input [15:0] A;
    input [3:0] r;
    output [15:0] W;

    assign W = A << r;
endmodule
```

FIGURA 115 – Implementació dels blocs Desplaçadors de 16 bits en Verilog.

MUX_2-1_1b

```
module mux2_1_1bit(a, b, sel, out);
  output out;
  input a,b,sel;

  not I5 (sel_n, sel);
  and I6 (sel_a, a, sel);
  and I7 (sel_b, b, sel_n);
  or I4 (out, sel_a, sel_b);
endmodule
```

sel	a	b	out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

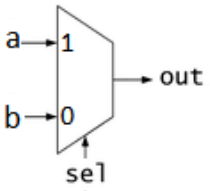


FIGURA 116 – Implementació del bloc Multiplexor de 2 entrades i 1 sortida de 1 bit en Verilog.

MULTIPLEXORS

```
module mux4_1_16bit(x3, x2, x1, x0, sel, out);
    output [15:0] out;
    input [15:0] x3,x2,x1,x0;
    input [1:0] sel;

    wire [15:0] out0, out1;

    mux2_1_16bit mux0 (x1, x0, sel[0], out0);
    mux2_1_16bit mux1 (x3, x2, sel[0], out1);
    mux2_1_16bit mux2 (out1, out0, sel[1], out);

endmodule

module mux8_1_16bit(x7, x6, x5, x4, x3, x2, x1, x0, sel, out);
    output [15:0] out;
    input [15:0] x7,x6,x5,x4,x3,x2,x1,x0;
    input [2:0] sel;

    wire [15:0] out0, out1;

    mux4_1_16bit mux0 (x3, x2, x1, x0, sel[1:0], out0);
    mux4_1_16bit mux1 (x7, x6, x5, x4, sel[1:0], out1);

    mux2_1_16bit mux2 (out1, out0, sel[2], out);

endmodule
```

FIGURA 117 – Implementació dels blocs Multiplexors de diverses entrades i 1 sortida de 16 bits en Verilog.

DEC_1-2_1b

```
module dec1_2(a,e,w1,w0);
  output w1,w0;
  input a;
  input e;
  wire a_n,e_n;

  not I0 (e_n, e);
  not I1 (a_n, a);

  and_1b and0 (a_n,e,w0);
  and_1b and1 (a[0],e,w1);

endmodule
```

e	a	w0	w1
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

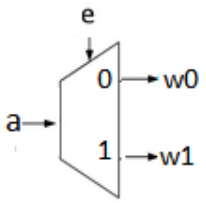


FIGURA 118 – Implementació del bloc Descodificador de 1 entrada i 2 sortides de 1 bit en Verilog.

DECODIFCADORS

```
module dec4_16(a,e,w15,w14,w13,w12,w11,w10,w9,w8,w7,w6,w5,w4,w3,w2,w1,w0);
  output w15,w14,w13,w12,w11,w10,w9,w8,w7,w6,w5,w4,w3,w2,w1,w0;
  input [3:0] a;
  input e;
  wire a3_n,a2_n,a1_n,a0_n,e_n;

  not I0 (e_n, e);
  not I1 (a0_n, a[0]);
  not I2 (a1_n, a[1]);
  not I3 (a2_n, a[2]);
  not I4 (a3_n, a[3]);

  and5_1b and0 (a0_n,a1_n,a2_n,a3_n,e,w0);
  and5_1b and1 (a[0],a1_n,a2_n,a3_n,e,w1);
  and5_1b and2 (a0_n,a[1],a2_n,a3_n,e,w2);
  and5_1b and3 (a[0],a[1],a2_n,a3_n,e,w3);
  and5_1b and4 (a0_n,a1_n,a[2],a3_n,e,w4);
  and5_1b and5 (a[0],a1_n,a[2],a3_n,e,w5);
  and5_1b and6 (a0_n,a[1],a[2],a3_n,e,w6);
  and5_1b and7 (a[0],a[1],a[2],a3_n,e,w7);
  and5_1b and8 (a0_n,a1_n,a2_n,a[3],e,w8);
  and5_1b and9 (a[0],a1_n,a2_n,a[3],e,w9);
  and5_1b and10 (a0_n,a[1],a2_n,a[3],e,w10);
  and5_1b and11 (a[0],a[1],a2_n,a[3],e,w11);
  and5_1b and12 (a0_n,a1_n,a[2],a[3],e,w12);
  and5_1b and13 (a[0],a1_n,a[2],a[3],e,w13);
  and5_1b and14 (a0_n,a[1],a[2],a[3],e,w14);
  and5_1b and15 (a[0],a[1],a[2],a[3],e,w15);
endmodule
```

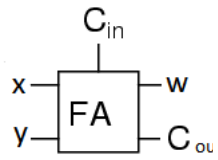
FIGURA 119 – Implementació del bloc Descodificador de 4 entrades i 16 sortides de 1 bit en Verilog.

FULL-ADDER

```
module fullAdder (x,y,c_in,w,c_out);
  input x;
  input y;
  input c_in;
  output w;
  output c_out;

  wire xXORy;
  wire c_inANDxXORy;
  wire xANDy;

  xor_1b XORgate1 (x,y,xXORy);
  xor_1b XORgate2 (xXORy,c_in,w);
  and_1b ANDgate1 (c_in,xXORy,c_inANDxXORy);
  and_1b ANDgate2 (x,y,xANDy);
  or_1b ORgate1 (c_inANDxXORy, xANDy, c_out);
endmodule
```



x	y	cin	cout	w
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

FIGURA 120 – Implementació del bloc Full-Adder en Verilog.

SUMADOR

```
module ADD_16b(A,B,Cin,W,Cout);
    input [3:0] A,B;
    input Cin;
    output [3:0] W;
    output Cout;

    wire c0,c1,c2;

    fullAdder bit0 (A[0],B[0],Cin,W[0],c0);
    fullAdder bit1 (A[1],B[1],c0,W[1],c1);
    fullAdder bit2 (A[2],B[2],c1,W[2],c2);
    fullAdder bit3 (A[3],B[3],c2,W[3],Cout);

endmodule
```

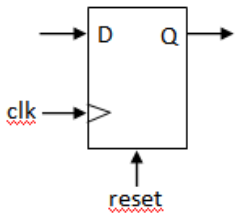
FIGURA 121 – Implementació del bloc Sumador de 16 bits en Verilog.

MÒDULS SEQÜENCIALS

A continuació trobem com implementar en Verilog els mòduls seqüencials que formen part dels components del CAL16.

BIESTABLE

```
module ff(q,clk,d,reset);
  output q;
  reg q;
  input clk,d,reset;
    always @ (posedge clk )
        if (reset)
            q <= 1'b0;
        else
            q <= d;
endmodule
```



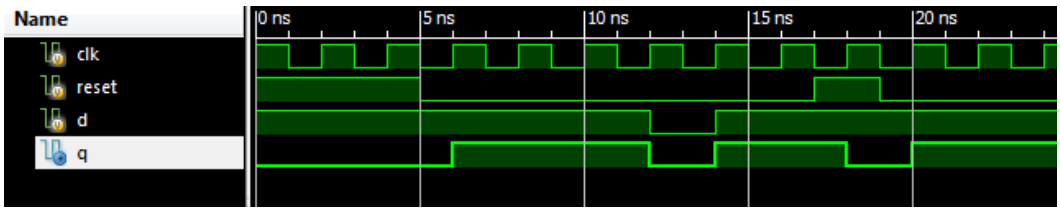


FIGURA 122 – Implementació del bloc Biestable de 1 bit en Verilog.

REGISTRE

```
module register_4bit (x,clk,w,reset);
    input [3:0] x;
    input clk,reset;
    output [3:0] w;

    ff bit0 (w[0],clk,x[0],reset);
    ff bit1 (w[1],clk,x[1],reset);
    ff bit2 (w[2],clk,x[2],reset);
    ff bit3 (w[3],clk,x[3],reset);

endmodule
```

FIGURA 123 – Implementació del bloc Registre de 4 bits en Verilog.

MANUAL D'USUARI

ÍNDEX

IMPLEMENTACIÓ DE PROJECTES EN VERILOG

CONÈIXER L'ENTORN

En aquest apartat es presenta l'entorn que ens proporciona *Xilinx ISE Design Suite* i la seva organització.

- Obrir el programa *Xilinx ISE Design Suite* fent clic sobre la icona: 
- El programa mostra la següent pantalla on:

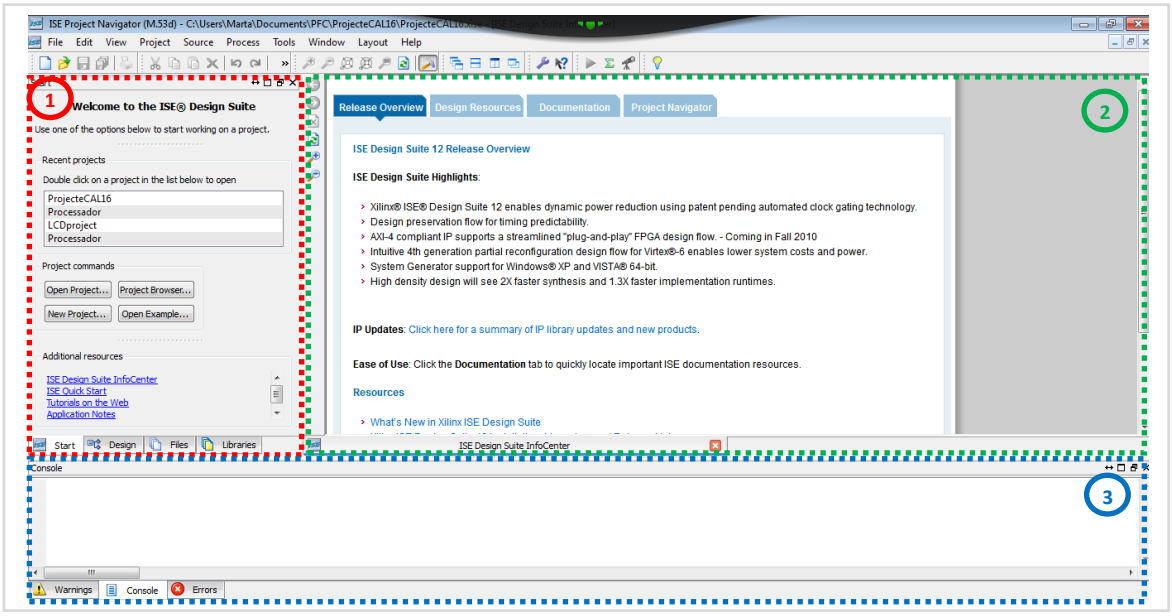


FIGURA 124 – Conèixer l'entorn, pantalla principal de l'eina Xilinx ISE Design Suite.

1. *Finestra d'informació del projecte*. Disposa de 4 pestanyes a escollir a la part inferior que permeten:

   - ***Start***: Iniciar el treball ja sigui obrint un projecte ja existent, com crear-ne un de nou.
   - ***Design***: Un cop obert o creat el projecte aquesta pestanya mostra en una finestra superior, anomenada *Hierarchy*, la organització dels seus continguts, permeten accedir als fitxers que el formen. En una finestra inferior, *Processes*, hi ha l'estat del projecte actual, on segons si et trobes en mode *Simulation* o *Implementation* (*View* ratio button superior) permet compilar, sintetitzar, simular o programar algun dispositiu amb el teu projecte.
   - ***Files***: Mostra el llistat de blocs continguts en el projecte amb les seves característiques.
   - ***Libraries***: Mostra el llistat de llibreries que formen part del projecte.

2. *Finestra d'edició del projecte*. En cas de voler realitzar algun canvi en algun fitxer o bloc, primer caldrà seleccionar-lo a la finestra de *Design* anterior, mitjançant un doble clic sobre ell, i es mostrarà el seu contingut en aquesta finestra. Per defecte inicialment al obrir un projecte en aquesta es mostra un resum sobre la informació del projecte.

3. *Finestra de control.* En aquest espai segons l'etiqueta que s'escull també poden visualitzar diferent informació sobre els processos que s'estan executant en el moment:
    - ***Warnings:*** En el cas de realitzar un procés de compilació, sintetització, implementació o generació del fitxer de programació d'un projecte, en aquesta es mostren el llistat de avisos que el procés genera.
    - ***Console:*** Mostra la informació de tots els passos que està portant a terme el procés que s'estigui executant en aquest moment.
    - ***Errors:*** En el cas de realitzar un procés de compilació, sintetització, implementació o generació del fitxer de programació d'un projecte, en aquesta es mostren el llistat de errors que el procés genera.

OBRIR UN PROJECTE

Per obrir un projecte ja existent seguirem els següents passos:

1. Obrir *Xilinx ISE Design Suite* fent clic a la icona que es troba en l'escriptori.
2. En la finestra d'informació del projecte escollir la pestanya *Start*. Clic sobre la opció *Open Project*, que obrirà la finestra per cerca la ubicació del projecte.

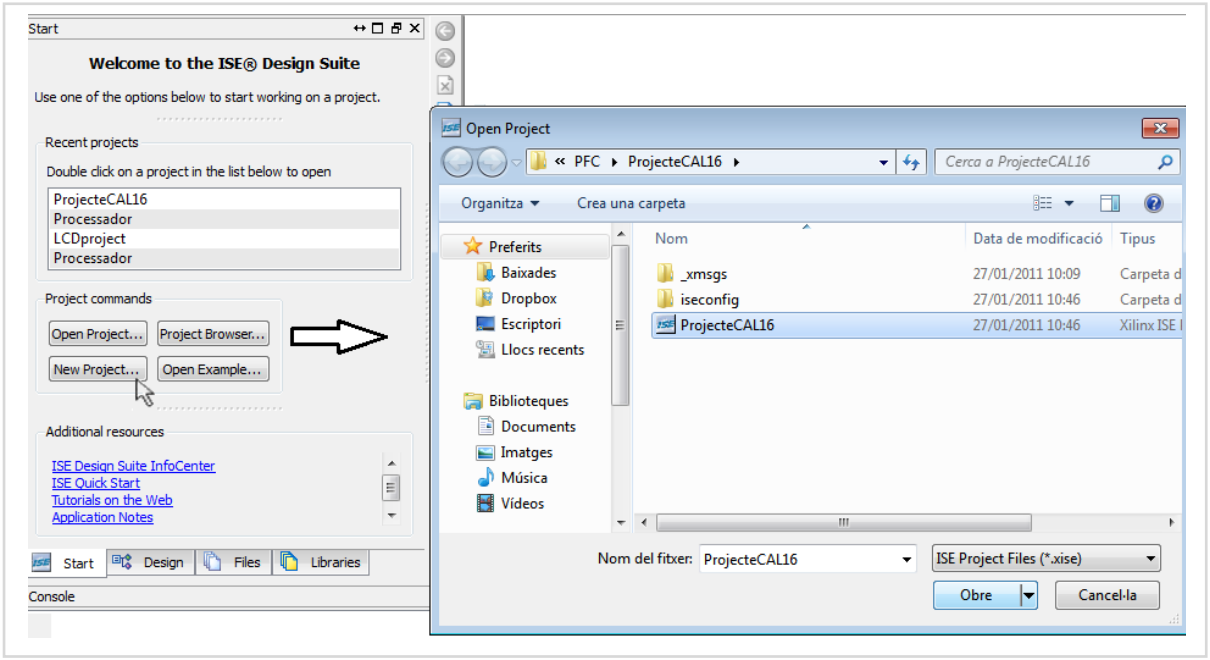


FIGURA 125 – Obrir un projecte, pas inicial.

3. Seleccionar-lo i clic sobre *Obre*.
4. Es mostrarà un fitxer resum del projecte escollit a la finestra d'edició. I a la pestanya de *Design* de la finestra d'informació del projecte tindrem la organització jeràrquica dels blocs que el formen, tal i com mostra la següent figura.

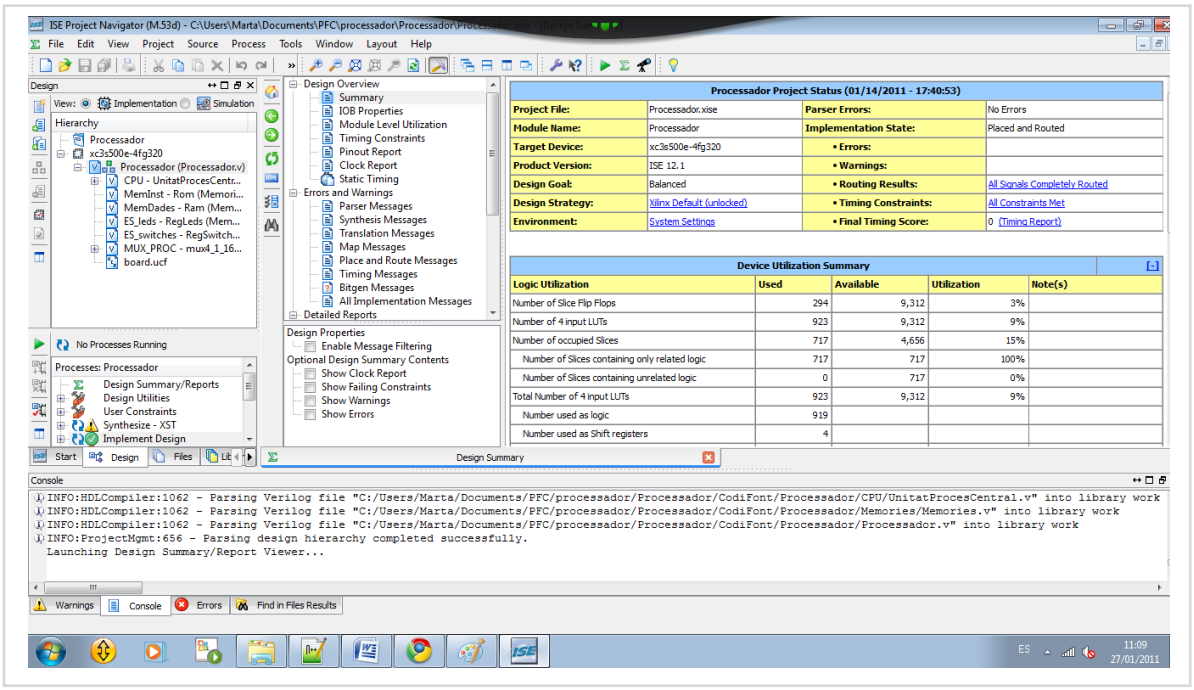


FIGURA 126 – Obrir un projecte, resum del projecte.

CREAR UN NOU PROJECTE

Per a crear un nou projecte en Verilog seguirem els següents passos:

1. Obrir *Xilinx ISE Design Suite* fent clic a la icona que es troba en l'escriptori.
2. En la finestra d'informació del projecte escollir la pestanya *Start*.
3. Fer clic sobre la opció *New Project*, que obrirà la finestra *New Project Wizard*.

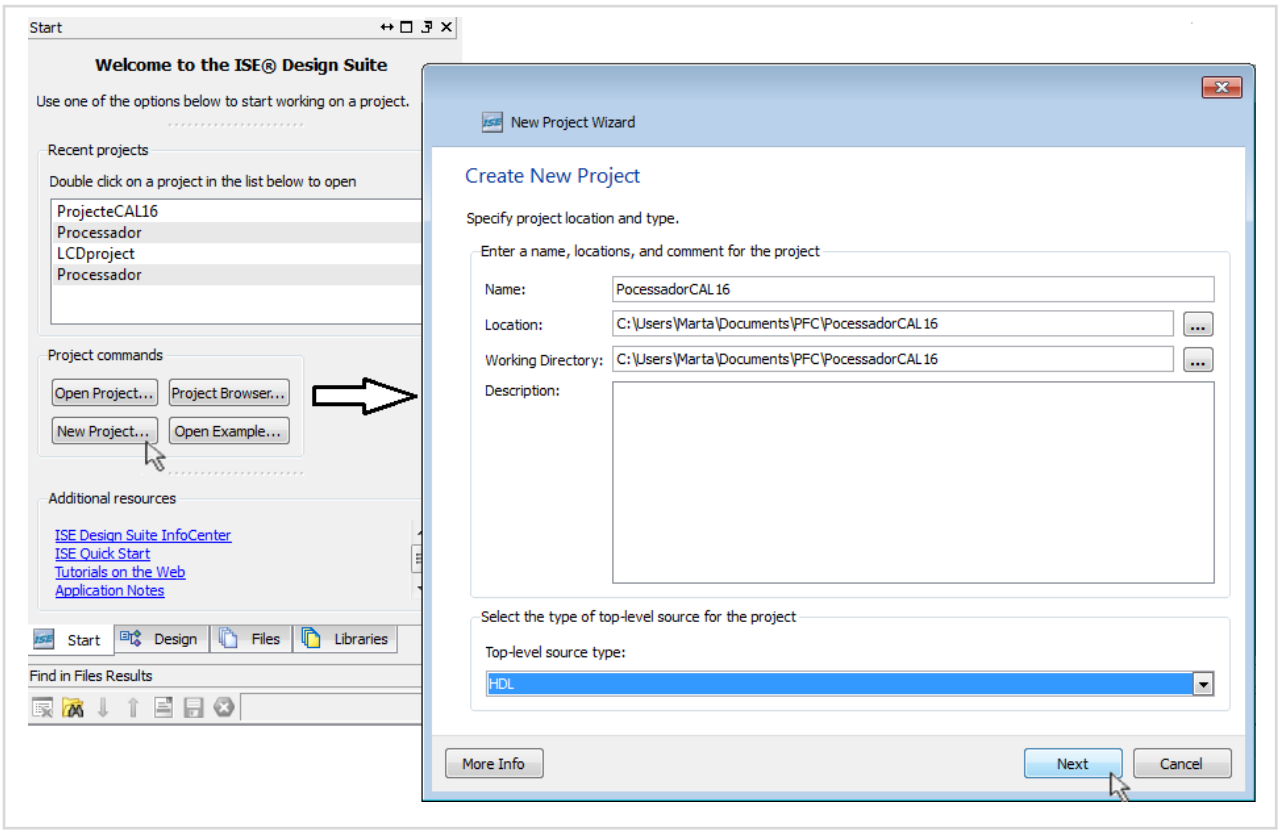


FIGURA 127 – Crear un nou projecte, pas inicial.

4. Omplir la informació del projecte (nom del projecte, directori de treball). Com a codi font escollir la opció HDL.
5. Clic sobre *Next*.

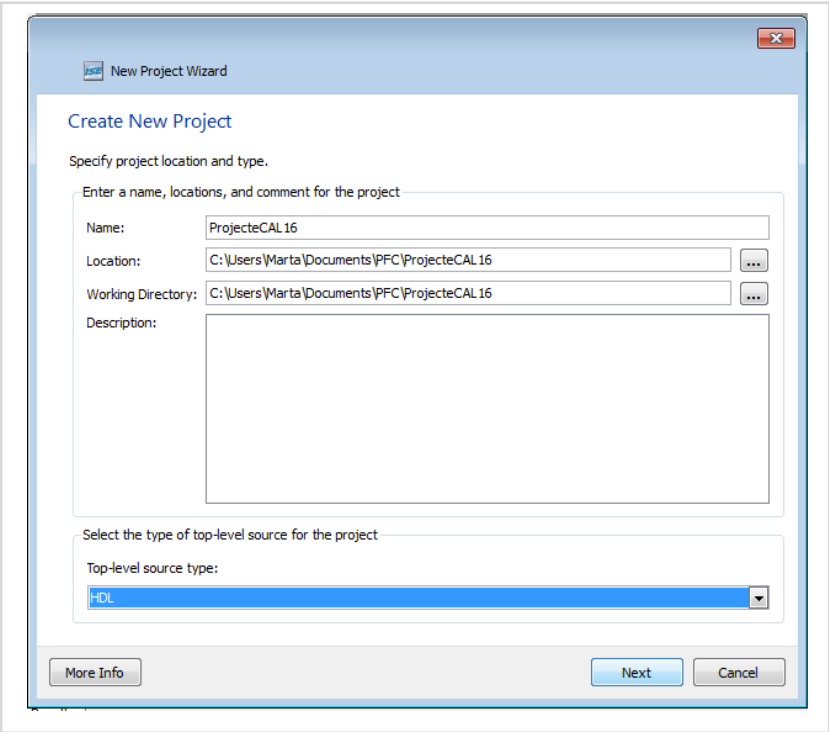


FIGURA 128 – Crear un nou projecte, determinar informació general.

6. Donar valor a les variables del projecte tal i com mostra la següent figura.
7. Clic *Next*.

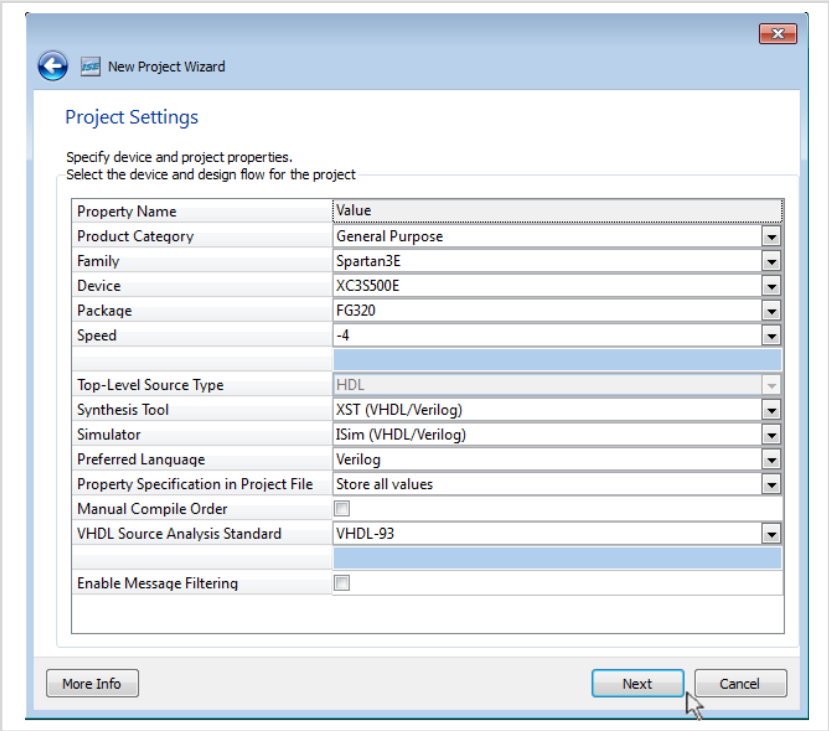


FIGURA 129 – Crear un nou projecte, donar valor a les variables.

8. Mostra la informació resum del nou projecte creat
9. Clic *Finish*.

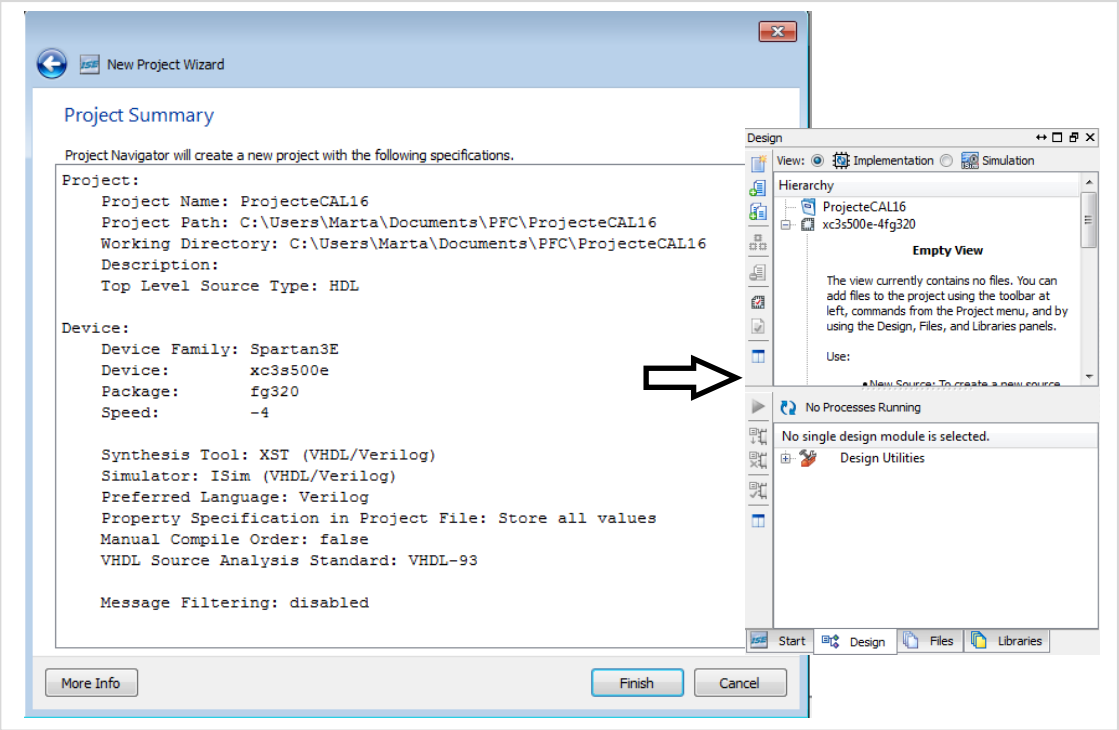


FIGURA 130 – Crear un nou projecte, confirmació i creació.

10. Ja tenim el nostre nou projecte creat, inicialment buit.

CREAR UN BLOC NOU

Per a crear un nou bloc en el nostre projecte tenim dues opcions:

- Modificar un fitxer que ja forma part del nostre projecte encapsulant el nostre bloc a continuació dels blocs ja implementats dins el fitxer.
- Crear un nou fitxer en Verilog per a implementar el nostre bloc.

Modificar un fitxer

Partim del punt en el que tenim el nostre projecte obert:

1. Seleccionar el fitxer al que volem afegir el bloc de la pestanya *Design* de la finestra de informació del projecte.
2. Doble clic sobre el fitxer.
3. A la pantalla d'edició del projecte s'haurà obert el codi font del fitxer seleccionat, editar-lo afegint el nostre bloc a continuació dels ja existents encapsulat dins les paraules

   Module <nomBloc> (<definició de busos d'entrada i de sortida>);
   <implementació>
   endmodule.

   Tal i com mostra el següent exemple d'un bloc afegit i ja implementat.

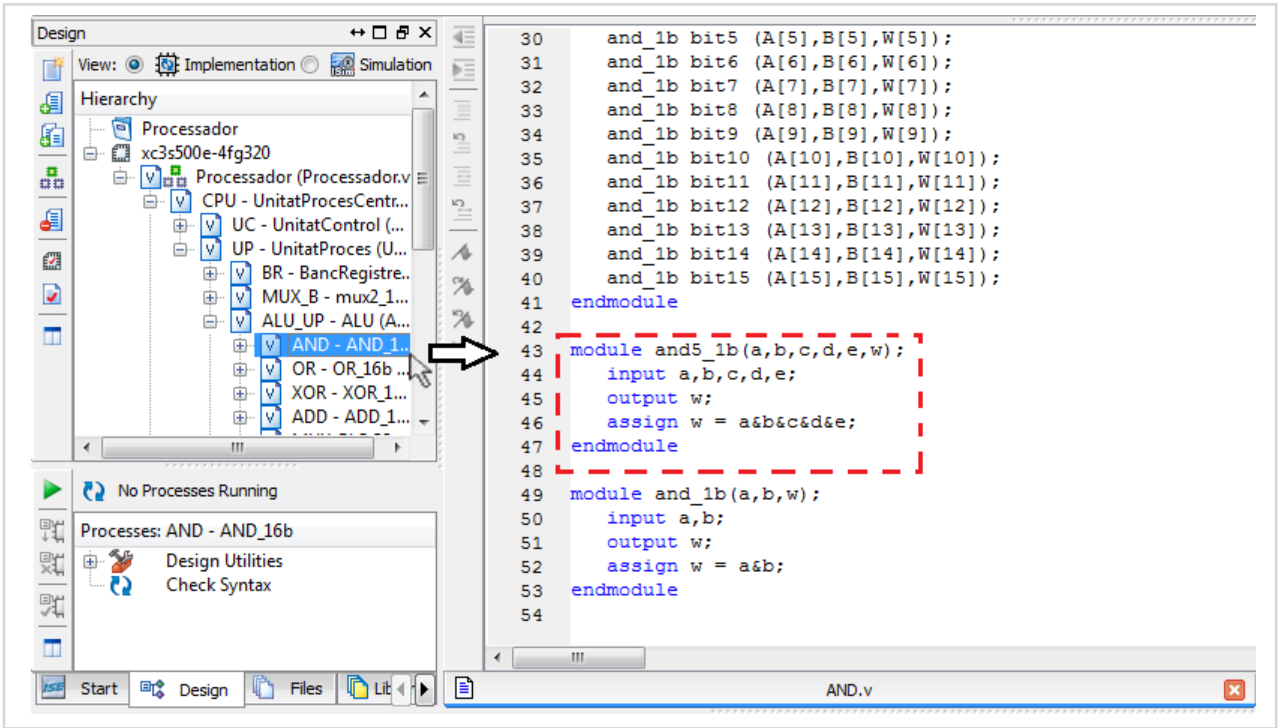
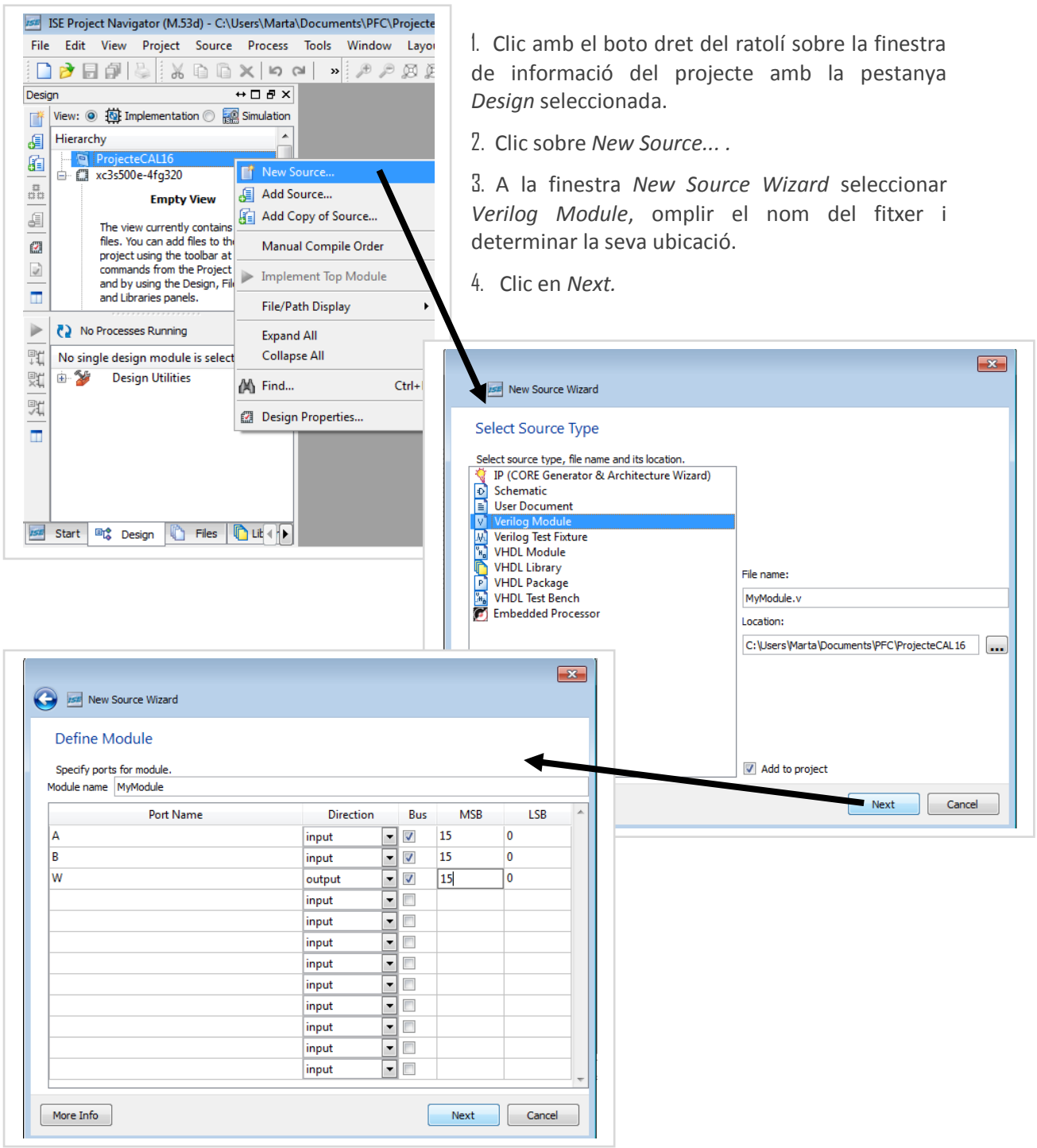


FIGURA 131 – Crear un nou mòdul. Dins un fitxer ja existent.

4. Ja podem començar a implementar el nou bloc.

Crear un fitxer

Partim del punt en el que tenim el nostre projecte creat:



1. Clic amb el boto dret del ratolí sobre la finestra de informació del projecte amb la pestanya *Design* seleccionada.
2. Clic sobre *New Source...* .
3. A la finestra *New Source Wizard* seleccionar *Verilog Module*, omplir el nom del fitxer i determinar la seva ubicació.
4. Clic en *Next.*

FIGURA 132 – Crear un nou mòdul. Creant un nou fitxer.

5. Es mostra una finestra anomenada *Define Module* que permet definir els busos d'entrada i de sortida del nostre bloc i que el mateix programa implementarà automàticament.
6. Clic en *Next.*

7. Es mostra una pantalla resum del bloc. Clic en *Finish*.

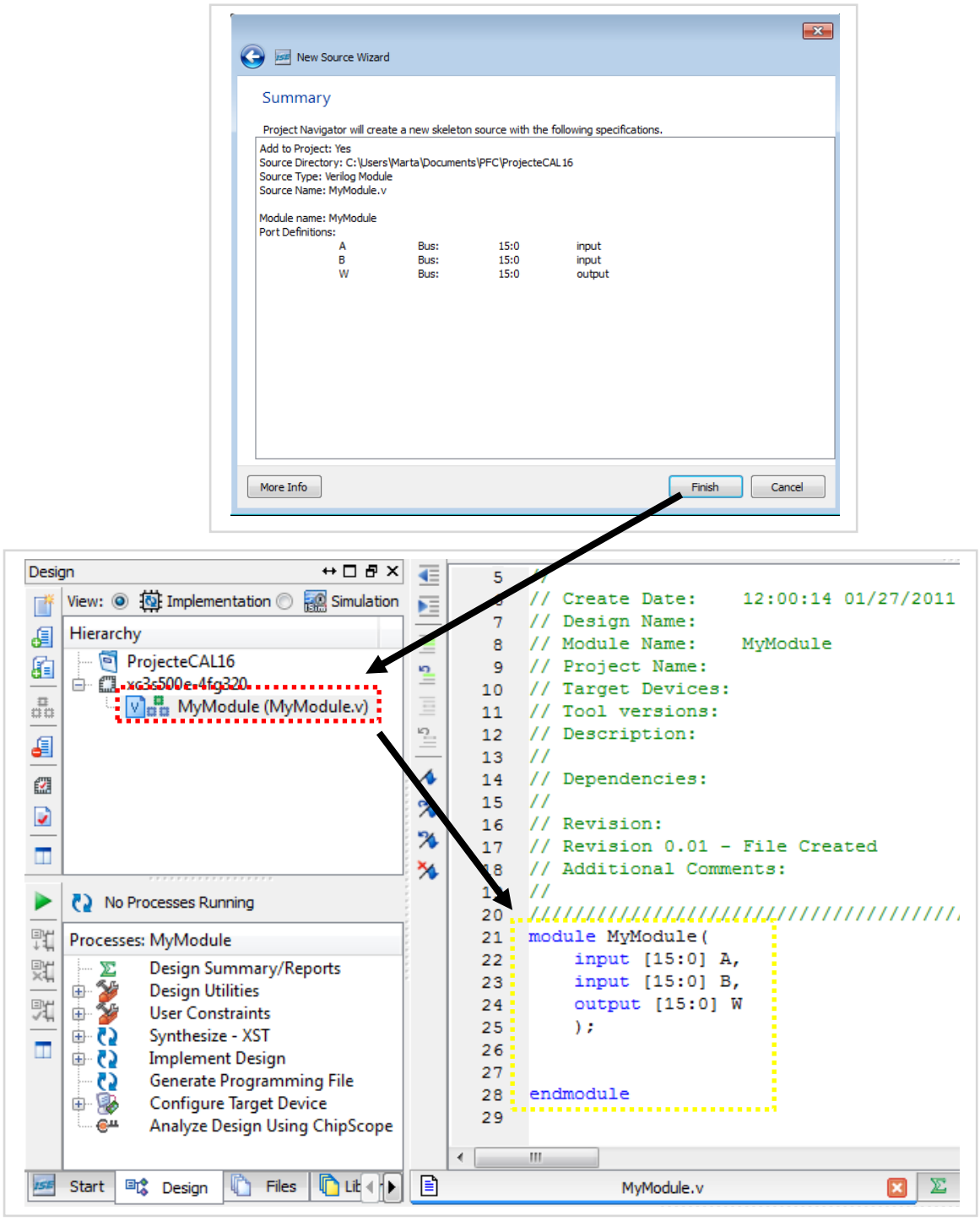


FIGURA 133 – Exemple de codi autogenerat en l'acció de crear un nou mòdul, creant un nou fitxer.

8. Ja podem començar a implementar el nou bloc.

EXECUTAR AMB ISE SIMULATOR

Un cop tenim el nostre projecte ja implementat, per a comprovar el seu correcte funcionament tenim varies opcions. Una d'elles consisteix en simular el seu comportament mitjançant l'eina ISE Simulator. En aquest apartat s'explica com fer-ho.

Primer de tot per a simular el comportament del bloc o conjunt de blocs que formen el nostre projecte, cal generar un fitxer de Test en el qual es poden definir els valors de les senyals o busos d'entrada en funció del temps. Un cop disposem d'aquest fitxer es pot passar a la pròpia simulació.

Crear un fitxer de Test

1. Seleccionar el projecte i clic amb el boto dret del ratolí.
2. Clic en *New Source... .*
3. S'obrirà una finestra anomenada *New Source Wizard* per seleccionar el tipus de fitxer a crear. Seleccionar *Verilog Test Fixure.* Donar-li nom al fitxer de test i si es vol canviar-li la ubicació per defecte.
4. Clic sobre *Next.*

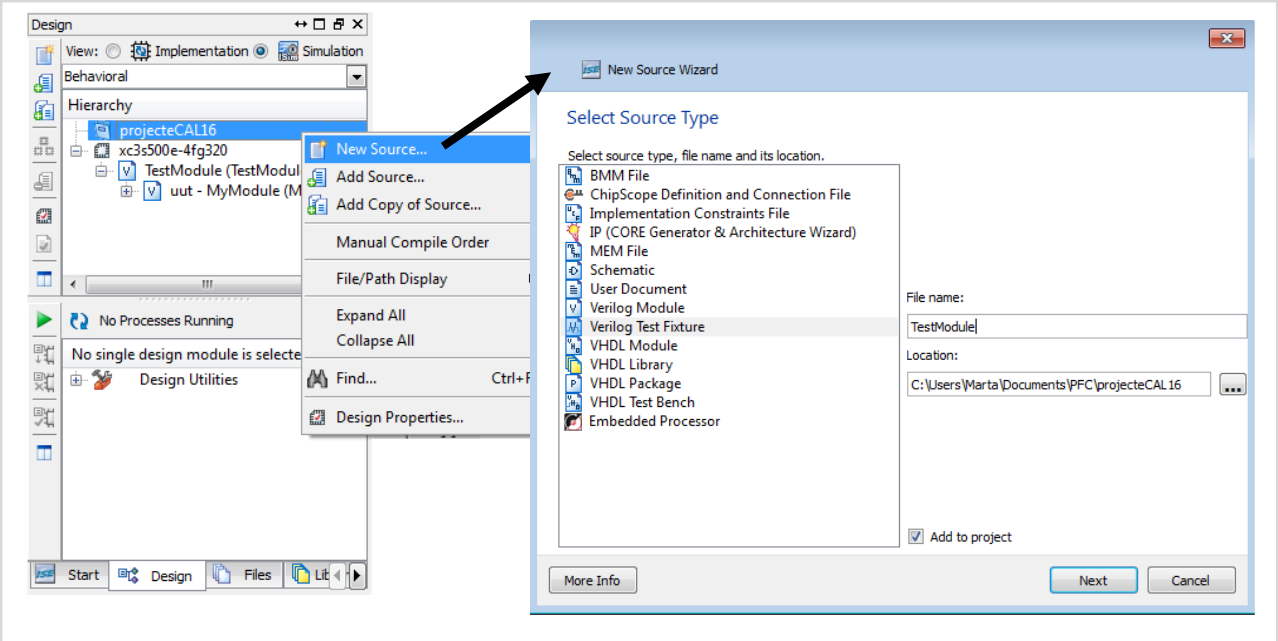


FIGURA 134 – Crear un fitxer de test per a la simulació amb l'eina ISim .

5. S'obre una finestra per seleccionar el bloc sobre el que vols generar el fitxer de test. Seleccionar-lo i clic en *Next*.
6. Es mostra una pantalla resum del fitxer a generar, clic *Finish*.

El fitxer de test generat automàticament per l'eina, té el següent aspecte:

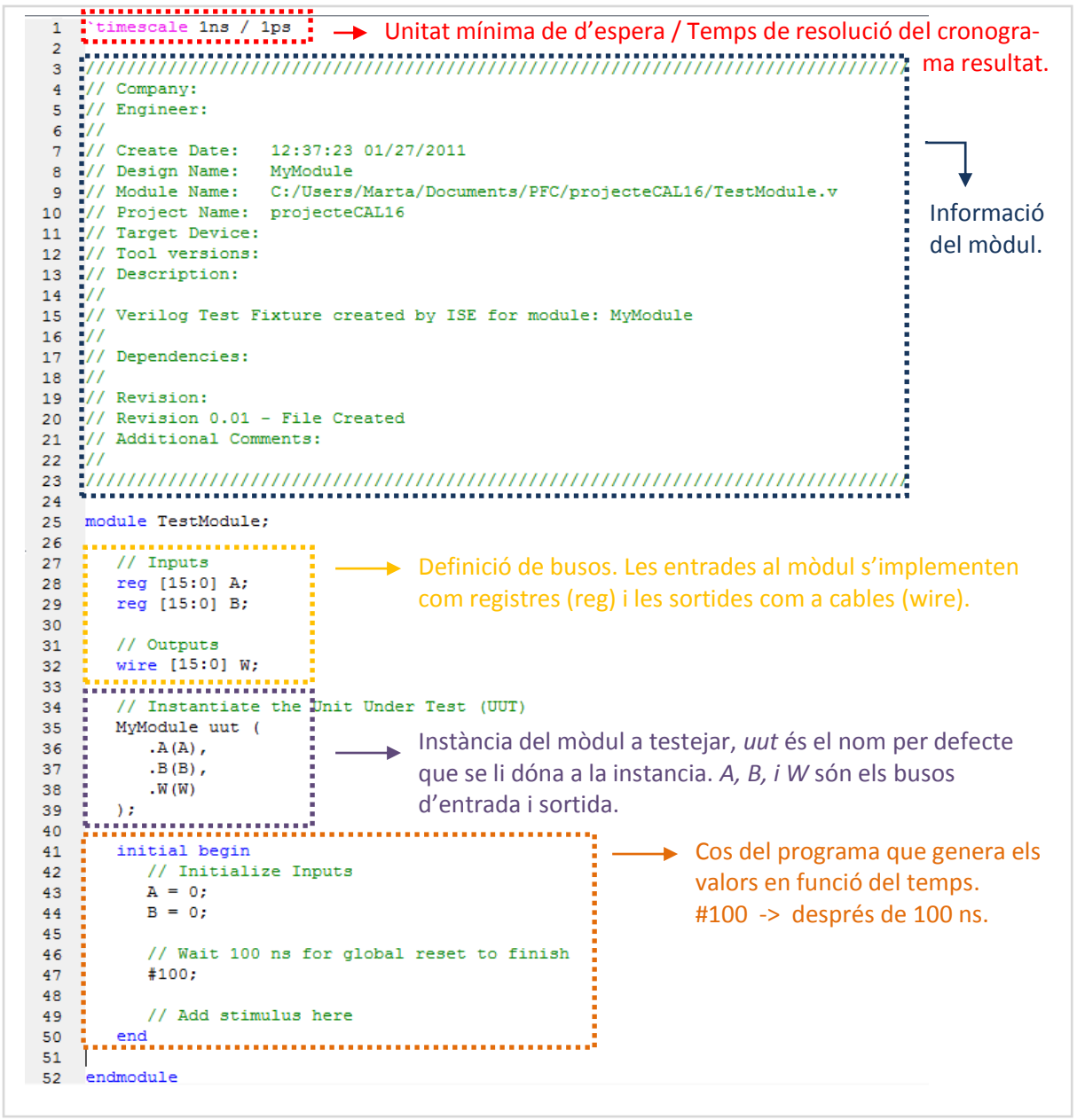


FIGURA 135 – Exemple del codi autogenerat al crear un fitxer de test per a la simulació amb l'eina ISim.

Un cop generat, haurem de donar valor als busos connectats a l'entrada del bloc, en funció del temps per a simular el seu comportament. A continuació es mostren dos exemples d'implementació del mòdul de test.

- Exemple **mòdul de test d'un bloc combinacional.**

```
module TestModule;

   // Inputs
   reg [15:0] A;
   reg [15:0] B;

   // Outputs
   wire [15:0] W;

   reg clk;
   // Instantiate the Unit Under Test (UUT)
   MyModule uut (
      .A(A),
      .B(B),
      .W(W)
   );

   initial begin
      A = 0;
      B = 0;
      #20 A <= 16'hFFFF;
      B <= 16'h0008;

      #20 A <= 16'h0003;
      B <= 16'hA5A5;

      #20 A <= 16'h9876;
      B <= 16'h1234;

   end
endmodule
```

Espera 20 ns.

FIGURA 137 – Exemple d'un mòdul de test pel bloc AND de 16 bits.

El mòdul **MyModule** instanciat en l'exemple anterior, respon al comportament d'una AND bit a bit, amb 2 busos de 16 bits d'entrada i un bus de 16 bits de sortida. A continuació es mostra el cronograma generat a partir dels valors del test:

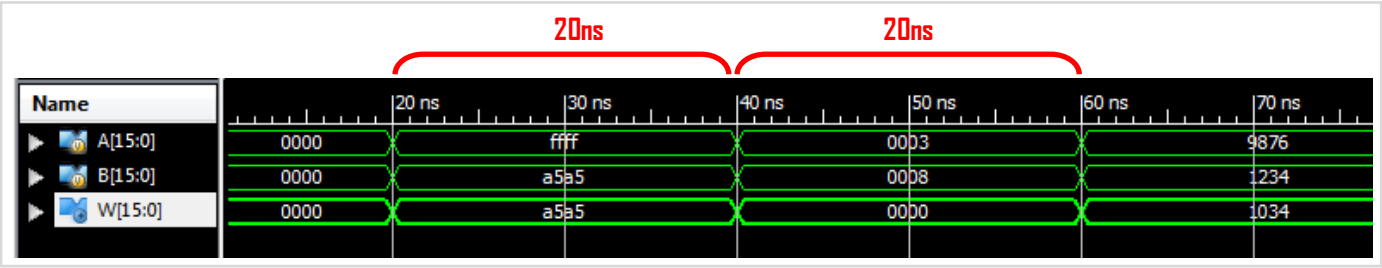


FIGURA 138 – Exemple del cronograma resultant de simular el comportament d'un mòdul de test pel bloc AND de 16 bits.

- Exemple fitxer de **test d'un bloc seqüencial.** Respon al comportament d'un Registre amb una entrada de 16 bits, una sortida de 16 bits, i senyals de càrrega (ld) i reinici (reset).

```
module TestModule;

   // Inputs
   reg [15:0] A;
   reg clk,reset,ld;

   // Outputs
   wire [15:0] W;


   register_16bit_ld REG16 (A,clk,W,reset,ld);

   initial begin
      // Initialize Inputs
      A <= 0;
      reset <= 1;
      ld <= 0;

      #10 reset <= 0;

      #10 A <= 16'hFFFF;
      #5  ld <= 1;
      #5  ld <= 0;

      #10 A <= 16'h0003;

      #10 A <= 16'h9876;
      #5  ld <= 1;

      #5  reset <= 1;
   end
   initial clk <= 1;
   always begin
      #1 clk <= ~clk;
   end

endmodule
```

Simulació senyal de rellotge.

FIGURA 139 – Exemple d'un mòdul de test pel bloc REG de 16 bits.

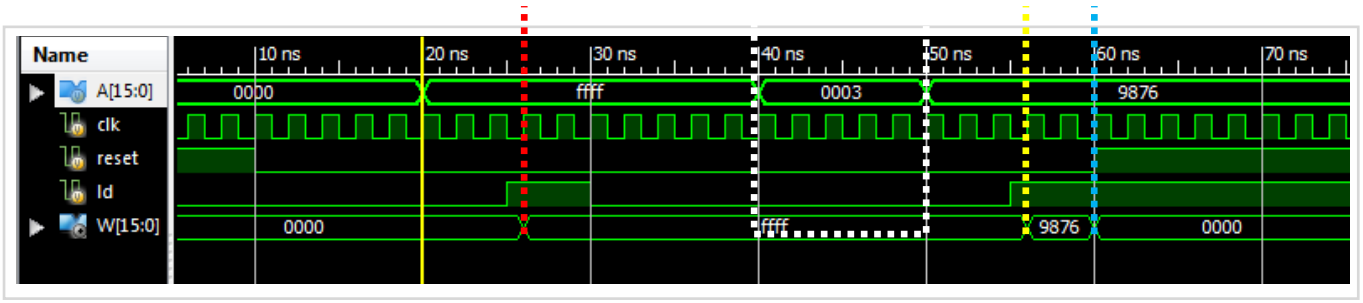


FIGURA 140 – Exemple del cronograma resultant de simular el comportament d'un mòdul de test pel bloc reg de 16 bits.

Simular l'execució del bloc

Un cop generat el fitxer de test, per a comprovar el correcte funcionament del bloc implementat, cal generar un cronograma com el mostrat en els exemples anteriors que mostra el valor dels senyals i busos i la seva evolució temporal.

Per a Simular el comportament del bloc cal:

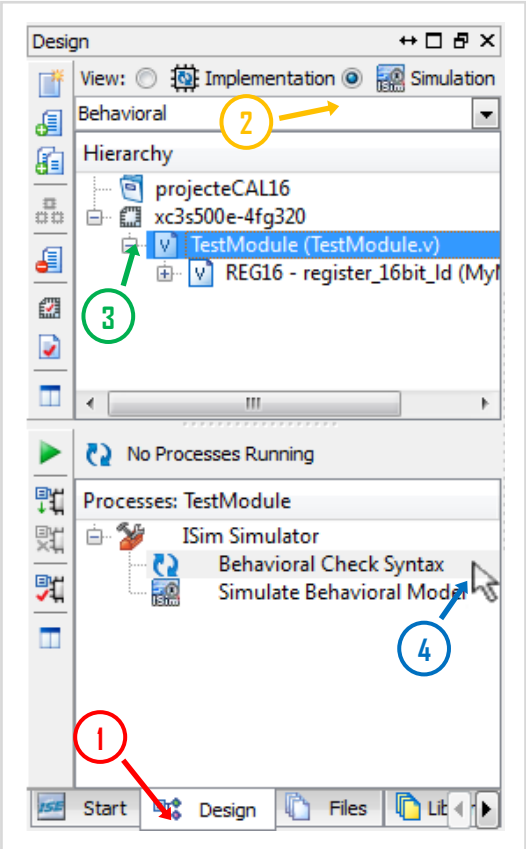


FIGURA 141 – Simular l'execució d'un bloc, pas inicial.

1. Seleccionar la pestanya *Design* de la finestra d'informació del projecte.

2. Marcar el ratio button de *Simulation* que es troba a la part superior de la finestra d'informació del projecte, anomenada *View*.

3. Seleccionar el fitxer de test que volem executar a la finestra superior anomenada *Hierarchy*.

4. Compilar-lo: Doble clic sobre *Behavioral Check Syntax,* a la finestra inferior anomenada *Processes*.

La compilació indicarà la seva correctesa mitjançant diferents icones i escrivint l'estat a la pestanya de *Console* de la finestra de control:

COMPILACIÓ INCORRECTE

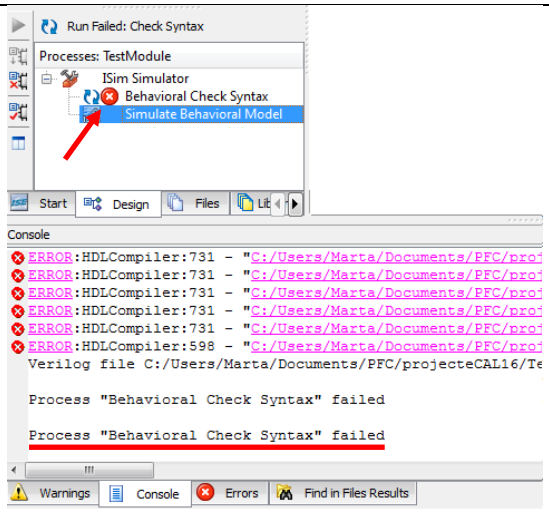


FIGURA 142 – Exemple de compilació del projecte incorrecte.

COMPILACIÓ CORRECTE

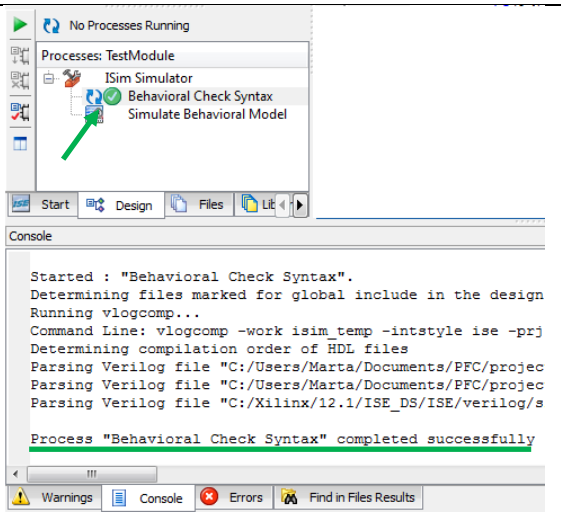
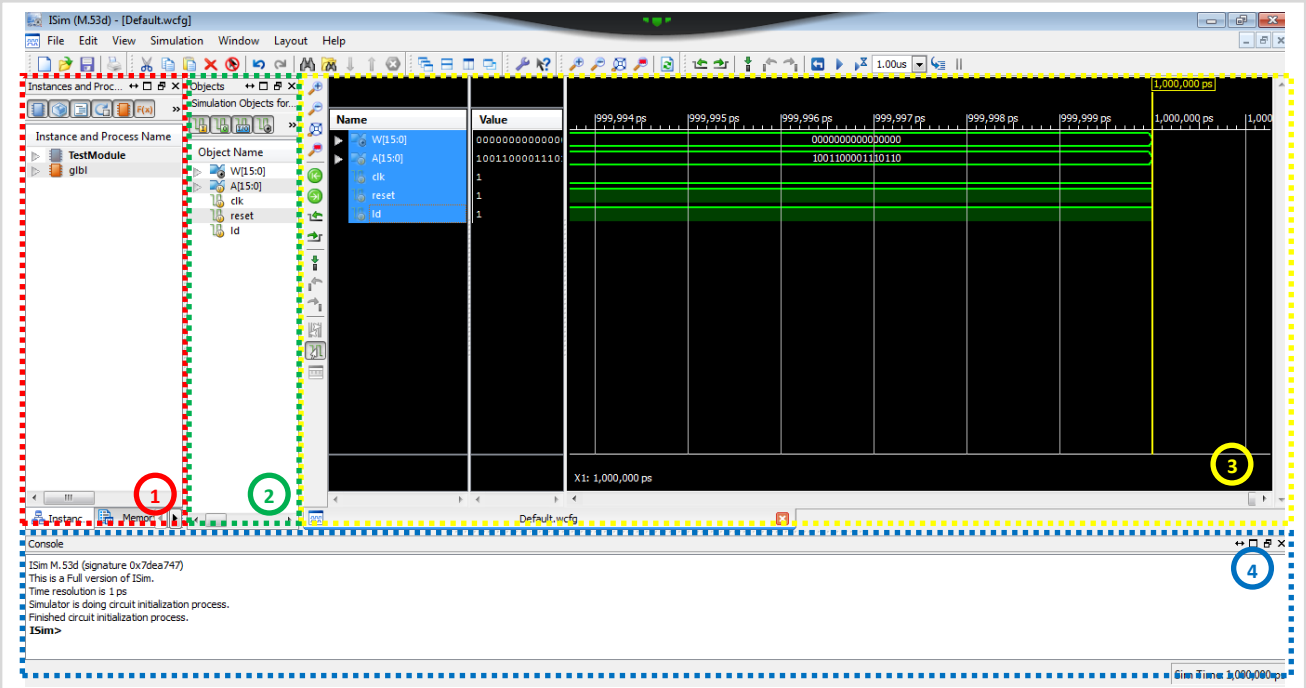


FIGURA 143 – Exemple de compilació del projecte correcte

En cas de donar errors en compilació els podem consultar tant a la pestanya de *Console* com a la de *Errors* de la Finestra de Control. Un cop arreglats tots el errors, quan disposem d'una compilació correcte, ja podem simular el seu comportament.

5. Doble clic sobre *Simulate Behavioral Model,* a la finestra anomenada *Processes*. Això obrirà l'eina de simulació *ISim* i ens mostrarà una pantalla com la següent:



1. *Finestra d'instàncies de la Simulació*. En aquesta fent doble clic sobre cadascun dels mòduls que formen el projecte podem anar accedint a les instancies que conté cada mòdul.
2. *Finestra d'objectes*. En funció de la instancia seleccionada en la finestra anterior, aquí es mostren els seu components, busos i senyals que formen el bloc seleccionat.
3. *Finestra de simulació*. Finestra on es genera el cronograma resultant d'executar el projecte.
4. *Finestra de consola.* En aquest espai mostra les comandes que s'executen en cada acció i l'estat de la simulació.

FIGURA 144 – Simular amb ISim, pantalla inicial.

EINA ISim

En aquest apartat ens centrarem en veure alguna funcionalitat útil per a treballar amb els cronogrames generats amb la simulació de projectes, tal i com hem vist a l'apartat anterior.

Un cop tenim l'eina ISim oberta, veiem quines funcions ens donen cadascuna de les finestres identificades en l'apartat anterior:

Finestra d'Instàncies i Finestra d'Objectes

Aquestes dues finestres estan molt relacionades entre si, ja que la Finestra d'Objectes mostra els components de la instància que tinguem seleccionada a la Finestra d'Instàncies.

FIGURA 145 – Eina ISim, finestra d'Objectes.

Fent clic amb el boto dret del ratolí sobre algun dels components s'obre un menú. Si escollim l'opció *Add To Wave Window* afegim el component al cronograma:

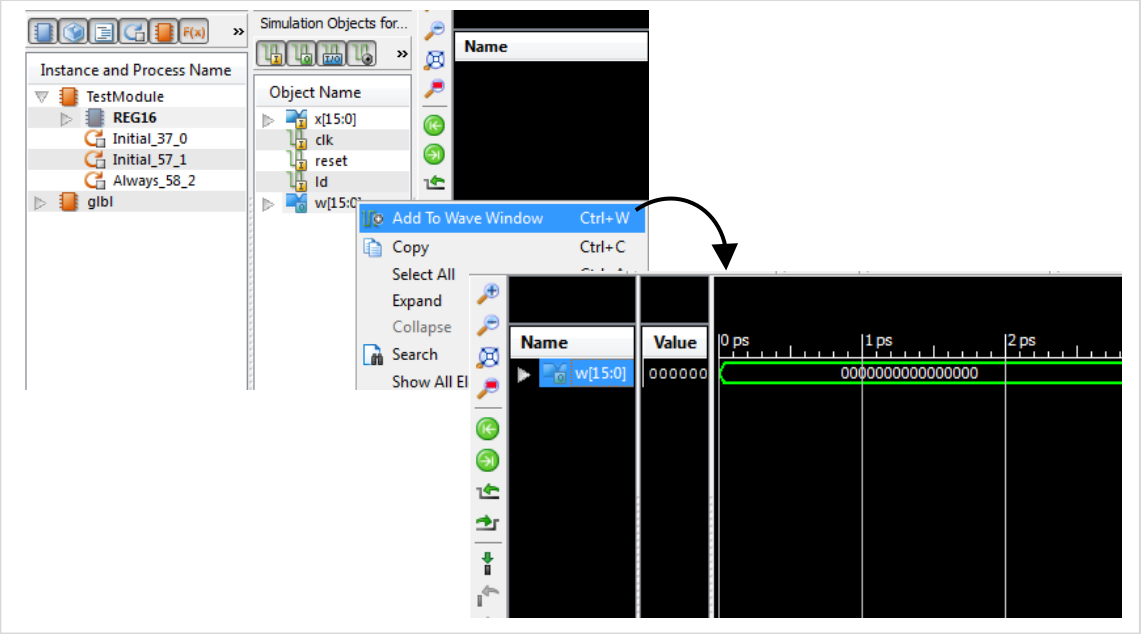


FIGURA 146 – Eina ISim, afegir un nou objecte al cronograma.

Amb el mateix procediment sobre la instancia afegim tots els seus components al cronograma.

Finestra de Simulació

En aquesta finestra hi ha dos menús d'icones al lateral esquerre i damunt del cronograma. Aquestes icones permeten manipular la visualització de la simulació.

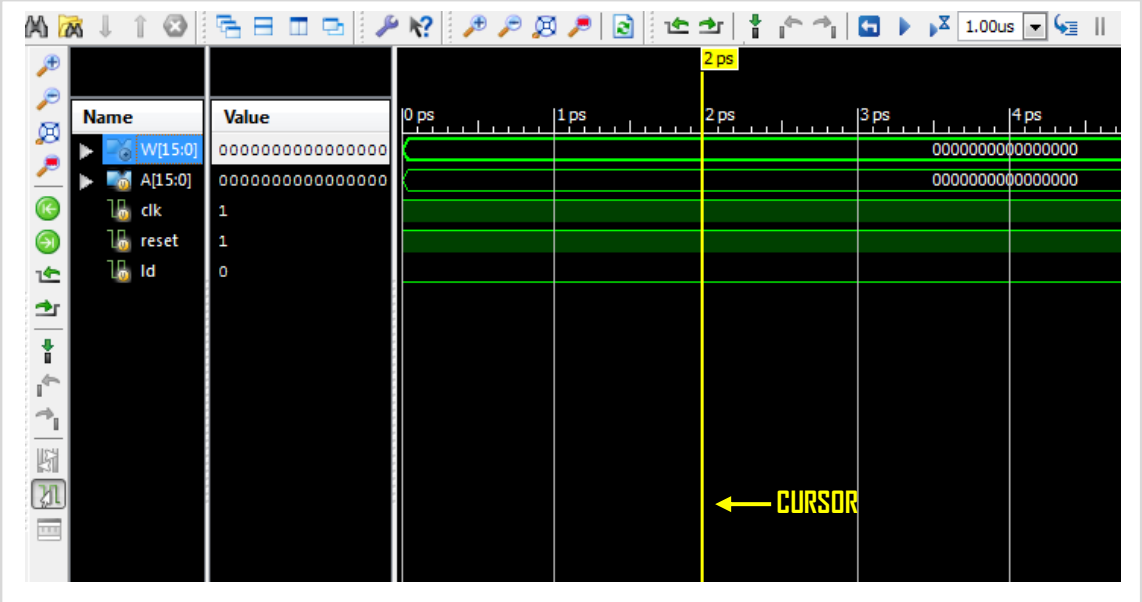


FIGURA 147 – Eina ISim, finestra de simulació.

 Veure cronograma amb interval de temps més ampli.

 Veure cronograma amb interval de temps més concret.

 Veure cronograma sencer, acotat a la mida de la finestra.

 Centrar el zoom en el cursor.

 Col·locar el cursor al inici de la simulació (temps = 0)

 Col·locar el cursor al final de la simulació.

 Avançar fins el següent canvi de valor del senyal/bus seleccionat.

 Retrocedir fins l'anterior canvi de valor del senyal/bus seleccionat.

 Afegir una marca (punt determinat dins la simulació).

 Retrocedir a la marca anterior.

 Avançar a la següent marca.

 Reinicia la simulació.

 Inicia la simulació.

Executa la simulació tantes unitats de temps com indica el desplegable.

 Pausa la simulació.

També podem determinar les propietats en funció de la senyal o bus. Si seleccionem un bus i fem clic amb el boto dret del ratolí, s'obre un menú que permet configurar la visualització del bus concret seleccionat. Aquest menú permet:

- Tallar un bus. [*Cut*]
- Copiar un bus. [Copy]
- Enganxar un bus. [*Paste*]
- Eliminar un bus del cronograma. [*Delete*]
- Reanomenar un bus. [Rename]
- Cercar en el llistat de busos. [*Find*]
- Seleccionar tots els busos. [*Select All*]
- Expandir un bus de més d'un bit per mostrar el valor de cadascun dels bits que el formen. [*Expand*]
- Recollir i ocultar el valor dels bits que formen un bus. [*Collapse*]
- Determinar si utilitzar el nom del bus simple [*Short*] o per contrari utilitzar com a nom del bus tota la URL que l'identifica [*Long*] en el llistat de busos. [*Name*]
- Determinar en quina base es mostra el valor del bus seleccionat, aquesta opció es mostra el seu efecte en la següent imatge. [*Radix*]
- Determinar el color del bus. [*Signal Color*]
- Intercanvia l'ordre dels bits que formen el valor del bus, és a dir el de més pes passa a ser a el de menys pes i viceversa per tots els bits. [*Reverse Bit Order*]
- Mostrar el codi que implementa el bloc al que pertany el bus en una nova pestanya fins aquesta mateixa finestra. [*Go To Source Code*]

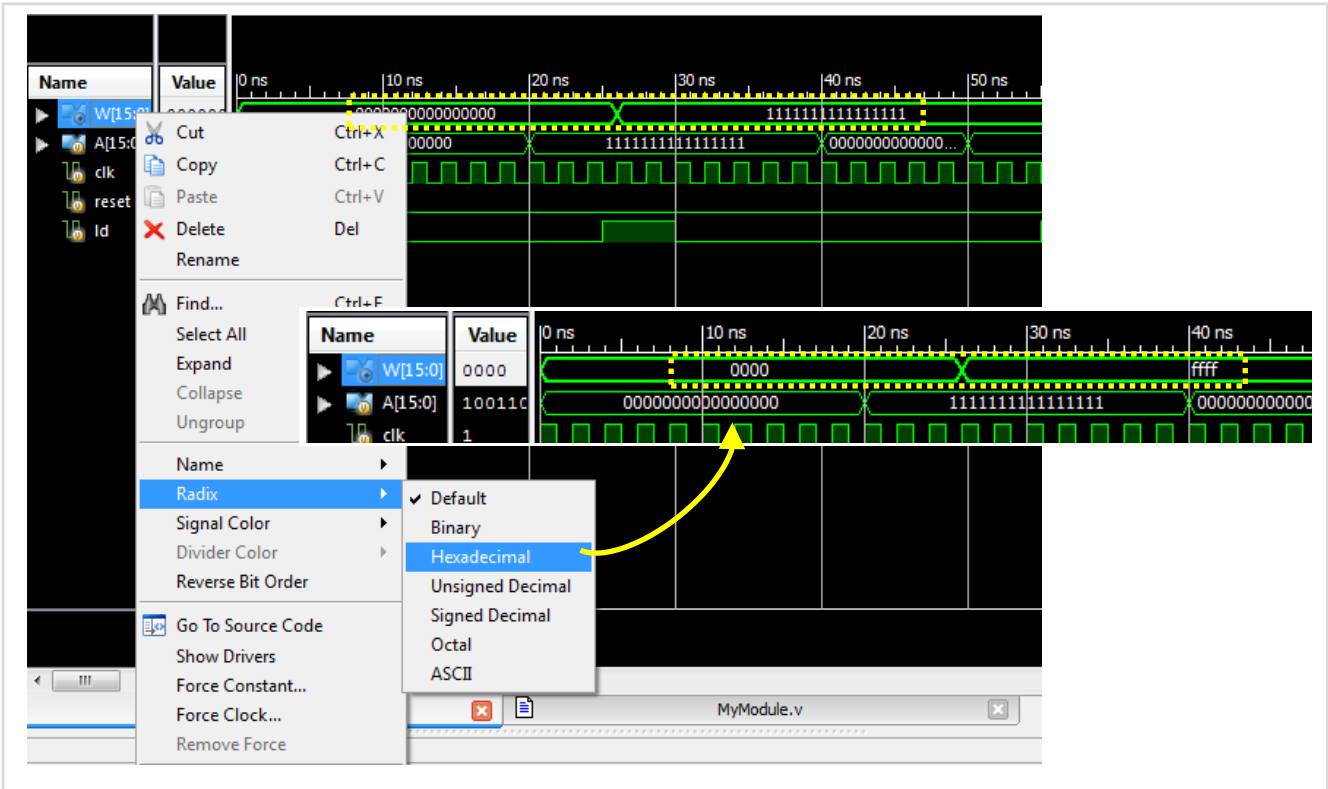


FIGURA 148 – Eina ISim, com modificar la base en que es mostren els valors dels busos del cronograma a la finestra de simulació.

PROGRAMAR UNA FPGA

Un cop disposem del projecte ja implementat, una altre opció, apart de la simulació, per a comprovar el seu correcte funcionament és programar una FPGA amb el codi del projecte i que aquesta es comporti tal i com li hem determinat.

Per a comprovar el seu correcte funcionament és necessari connectar els dispositius d'entrada sortida de la FPGA amb els senyals d'entrada sortida del bloc. En aquest cas doncs, per començar cal afegir aquest fitxer de configuració al projecte, per a que identifiqui cadascun dels dispositius dels que disposa la placa.

Crear un fitxer de configuració

1. Marcar el radio button *Simulation* que es troba en la part superior de la pestanya *Design* de la finestra d'informació del projecte.
2. Seleccionar el projecte i clic amb el boto dret del ratolí.
3. Clic en *New Source... .*
4. S'obrirà una finestra anomenada *New Source Wizard* per seleccionar el tipus de fitxer a crear. Seleccionar *Implementation Constraint File.* Donar-li nom al fitxer de configuracio <*nom.ucf*> i si es vol, canviar-li la ubicació per defecte.
5. Clic sobre *Next.*

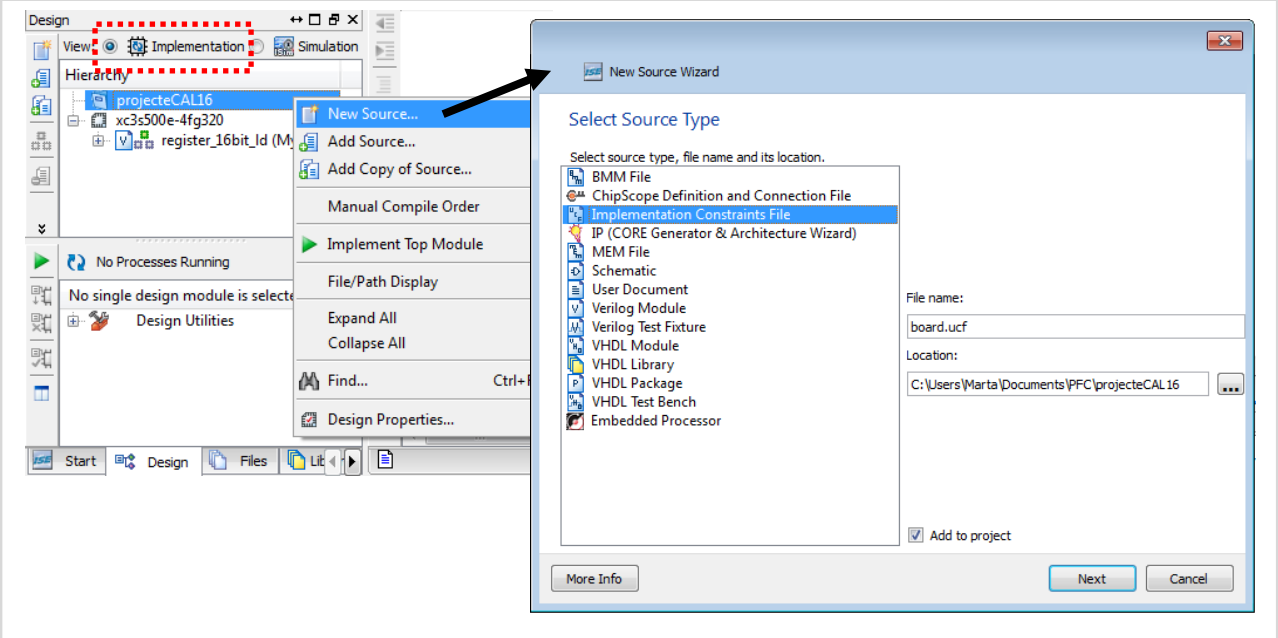


FIGURA 149 – Programar un FPGA, crear un fitxer de configuració.

6. A continuació s'obrirà una finestra amb el resum de les propietats del fitxer que acabem de crear. Clic *Finish*.

Aquest procés, haurà creat automàticament un fitxer en blanc com a part del nostre bloc principal del projecte.

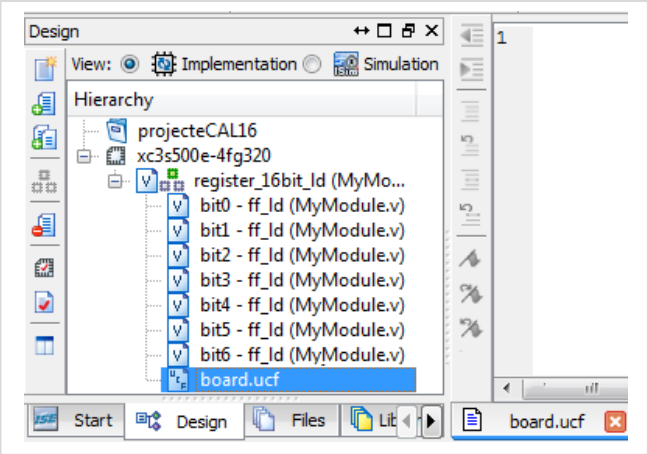


FIGURA 150 – Programar un FPGA, col·locació del fitxer de configuració.

7. Per implementar-lo cal afegir la declaració dels dispositius que es vulguin utilitzar en el programa. A continuació nomes es mostra com identificar aquells que utilitza el CAL16[1].

```
#############################################################
### SPARTAN-3E STARTER KIT BOARD CONSTRAINTS FILE ######
#############################################################

# ==== Pushbuttons (BTN) ====
#NET "BTN_EAST"  LOC = "H13" | IOSTANDARD = LVTTL  | PULLDOWN ;
NET "BTN_NORTH" LOC = "V4"  | IOSTANDARD = LVTTL  | PULLDOWN ;
NET "BTN_SOUTH" LOC = "K17" | IOSTANDARD = LVTTL  | PULLDOWN ;
#NET "BTN_WEST"  LOC = "D18" | IOSTANDARD = LVTTL  | PULLDOWN ;
#NET "CLK_SMA"   LOC = "A10" | IOSTANDARD = LVCMOS33 ;

# ==== Clock inputs (CLK) ====
NET "clk" LOC = "C9" | IOSTANDARD = LVTTL;
#     Define clock period for 50 MHz oscillator (40%/60% duty-cycle)
NET "clk" PERIOD = 20.0ns HIGH 40%;
#NET "CLK_AUX"   LOC = "B8"  | IOSTANDARD = LVCMOS33 ;
#NET "CLK_SMA"   LOC = "A10" | IOSTANDARD = LVCMOS33 ;

# ==== Slide Switches (SW) ====
NET "SW<0>"  LOC = "L13" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<1>"  LOC = "L14" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<2>"  LOC = "H18" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<3>"  LOC = "N17" | IOSTANDARD = LVTTL | PULLUP ;
```

FIGURA 151 – Fitxer de configuració boad.ucf un cop descomentats els dispositius de la FPGA Spartan 3E necessàris.

[1] *Board.ucf: Per a obtenir el fitxer board.ucf amb tots els dispositius disponibles a la FPGA Spartan 3E comentats, veure CD adjunt en el capítol d'Annexes, Annes 3, dins el directori de Recursos Exercicis\ el fitxer s'anomena white-board.ucf.*

```
# ==== Discrete LEDs (LED) ====
# These are shared connections with the FX2 connector
NET "LED<0>"  LOC = "F12" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<1>"  LOC = "E12" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<2>"  LOC = "E11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<3>"  LOC = "F11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<4>"  LOC = "C11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<5>"  LOC = "D11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<6>"  LOC = "E9"  | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<7>"  LOC = "F9"  | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
```

FIGURA 151 – Fitxer de configuració boad.ucf un cop descomentats els dispositius de la FPGA Spartan 3E necessàris.

En el fitxer anterior es mostra en verd els comentaris i en negre les declaracions dels dispositius a utilitzar, més concretament a la FPGA Spartan 3E disposem de:

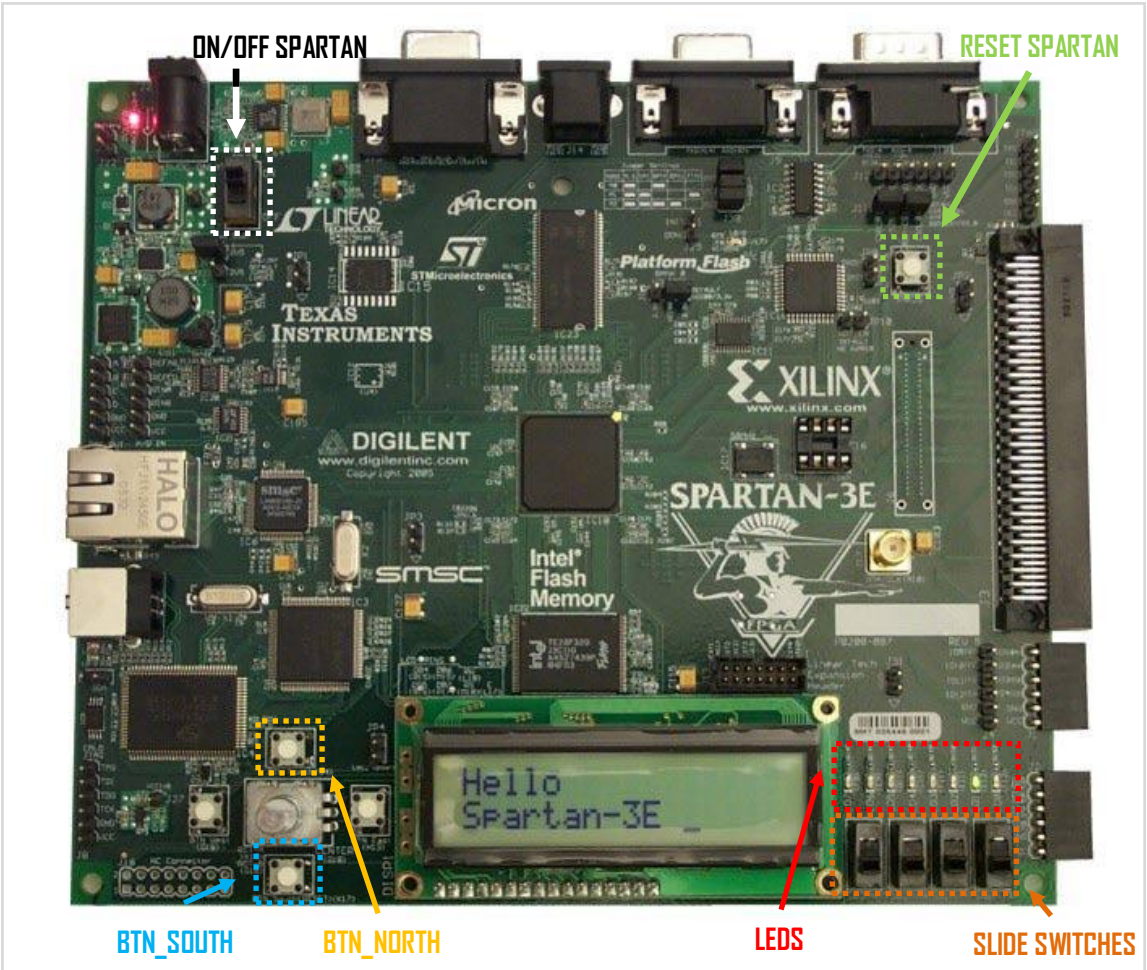


FIGURA 152 – Identificació dels dispositius utilitzats en projecte sobre la FPGA Spartan 3E.

8. Implementar l'ús dels dispositius de la PFGA en el projecte. Cal programar-los d'entrada/sortida del bloc principal del nostre projecte utilitzant sempre el mateix nom amb el que s'han definit al fitxer de configuració. A continuació es mostra un exemple de com programar un bloc que tingui d'entrada el rellotge, els interruptors i els botons North i South de la FPGA i de sortida el leds.

```
module MainModule(input clk, input [3:0] sw, input BTN_SOUTH, input BTN_NORTH,
                  output [7:0] led);

  wire [15:0] w;

  assign led = w[7:0];

  register_16bit_ld REG16 ({12'h000,sw},clk,w,BTN_SOUTH,BTN_NORTH);
  //BTN_SOUTH -> reset
  //BTN_NORTH -> ld

endmodule
```

FIGURA 153 – Exemple de connexió entre dispositius de la FPGA Spartan 3E i l'entrada i sortida del bloc REG de 16 bits..

9. Guardar el fitxer de configuració modificat. Clic en guardar.

Programar la FPGA amb el projecte

1. Seleccionar el mòdul arrel (principal) del nostre projecte, a la finestra d'informació del projecte, a la pestanya de *Design,* a la subfinestra superior anomenada *Hierarchy*.
2. Dins la finestra d'informació del projecte, a la pestanya de *Design,* a la subfinestra inferior anomenada *Processes*, doble clic sobre l'opció *Synthesize XST*.

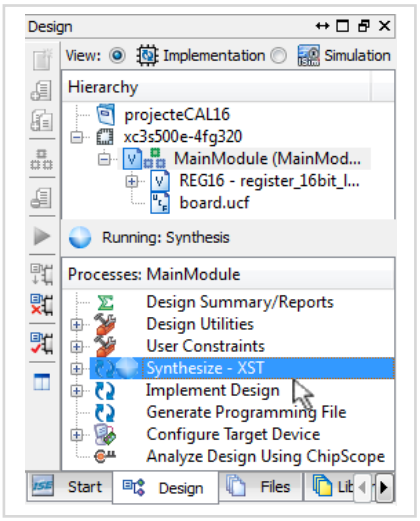


FIGURA 154 – Programar una FPGA, sintetitzar.

3. Si hi ha errors de sintetització cal arreglar-los per continuar. Detectarem la correctesa per que ens mostrarà un icona com els següents al costat de l'opció:

Synthesize - XST	Sintetització correcte.
Synthesize - XST	Sintetització correcte però amb avisos de possibles errors.
Synthesize - XST	Sintetització incorrecte, cal arreglar els errors mirant a la finestra de control, la pestanya *Errors*.

FIGURA 155 – Programar una FPGA, opcions de correctesa en la sintetització.

4. Doble clic sobre l'opció *Implement Design*.

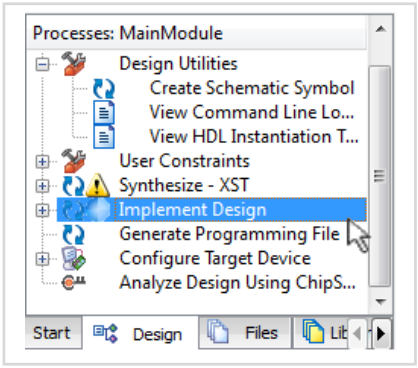


FIGURA 156 – Programar una FPGA, implementar el disseny.

5. Si hi ha errors de implementació cal arreglar-los per continuar. Detectarem la correctesa per que ens mostrarà un icona al costat de l'opció com les explicades anteriorment en el punt 3.

6. Doble clic sobre l'opció *Generate Programming File*.

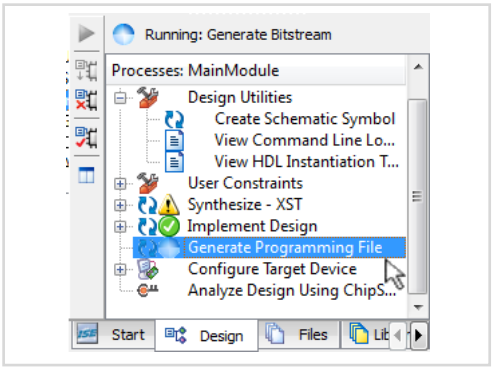


FIGURA 157 – Programar una FPGA, generar el fitxer de programació.

7. Si hi ha errors de generació del fitxer de programació cal arreglar-los per continuar. Detectarem la correctesa per que ens mostrarà un icona al costat de l'opció com les explicades anteriorment en el punt 3.

8. Doble clic sobre l'opció *Configure Target Device.*

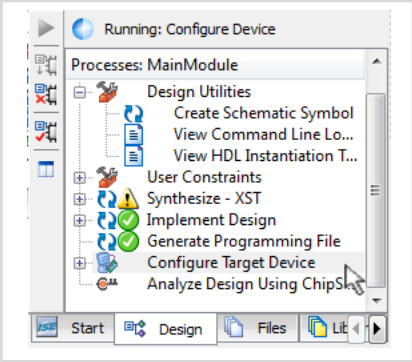


FIGURA 158 – Programar una FPGA, pas final.

9. Endollar la FPGA Spartan 3E al corrent elèctric i connectar-la a través dun port USB a l'ordinador.

10. En aquest punt hi ha dos camins possibles:

- Si ja s'ha configurat el projecte algun cop per a programar una FPGA, aquest ja té guardades les opcions de configuració. Passar directament el pas 32.
- En cas que sigui el primer cop que el volem programar a una FPGA, caldrà indicar-li les configuracions. Continuar en el punt 9.

11. En cas de tractar-se del primer cop que preparem el projecte per a programar la Spartan 3E, se'ns obrirà la següent pantalla:

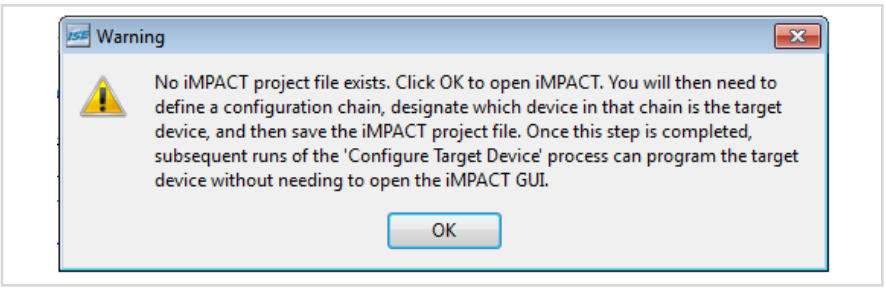


FIGURA 159 – Programar una FPGA, avís conforme el projecte no disposa de configuracions de programació i cal determinar-les.

12. Clic *OK*.

13. S'obrirà una eina anomenada *ISE iMPACT* en una nova finestra.
14. Doble clic sobre *Boundary Scan*.
15. Clic amb el boto dret sobre la finestra on s'ha obert el *Boundary Scan*.
16. S'obrirà un menú desplegable, clic sobre *Cable Auto Connect.*
17. Clic amb el boto dret sobre la finestra on s'ha obert el *Boundary Scan*.
18. S'obrirà un menú desplegable, clic sobre *Initialize Chain.*

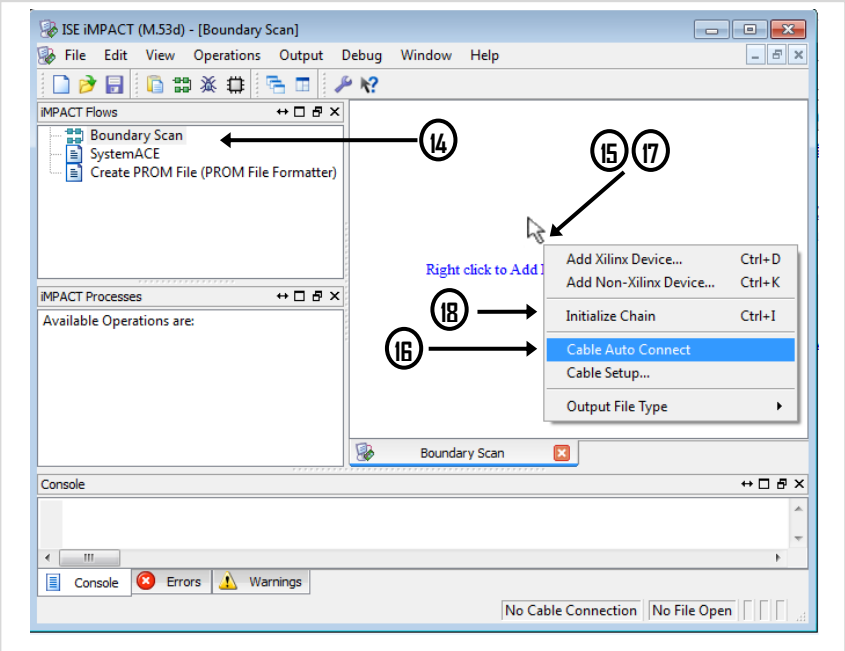


FIGURA 160 – Programar una FPGA, determinar les configuracions de programació.

19. Apareixerà un figura com la següent, en la finestra de *Boundary Scan*. Clic amb el botó dret sobre el primer component, i en el menú desplegable clic sobre *Assign New Configuration File.*

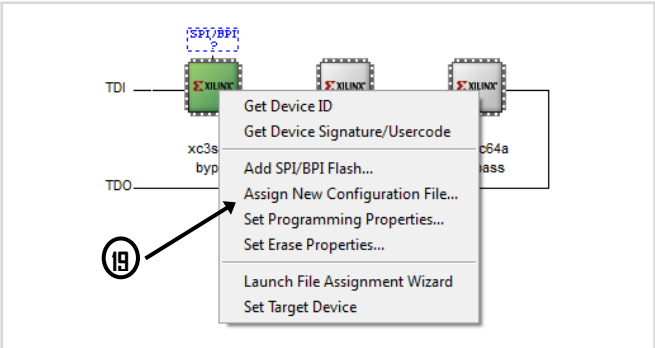


FIGURA 161 – Programar una FPGA, assignar les configuracions de programació a un fitxer.

20. S'obrirà una finestra on buscar el mòdul principal del projecte. Seleccionar-lo i clic en *Obre.*

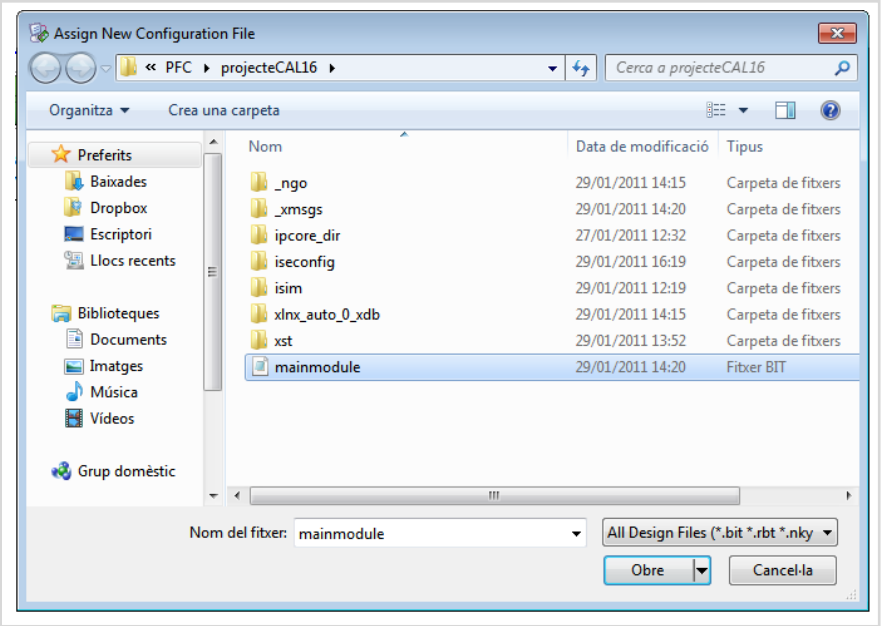


FIGURA 162 – Programar una FPGA, assignar les configuracions de programació al fitxer amb el que vull programar la FPGA.

21. Es crearà un nou fitxer anomenat <nom_modul_principal>.bit, ho podrem identificar en la següent pantalla.

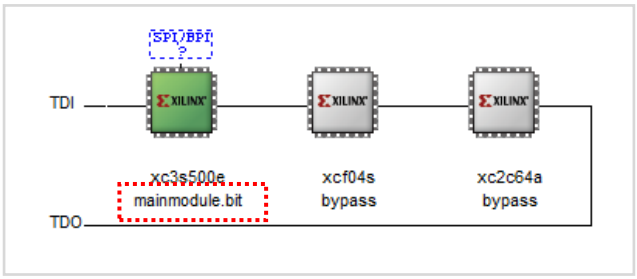


FIGURA 163 – Programar una FPGA, comprovar que el fitxer ha estat correctament assignat.

22. Clic amb el botó dret sobre el primer component, i en el menú desplegable clic sobre *Program.*

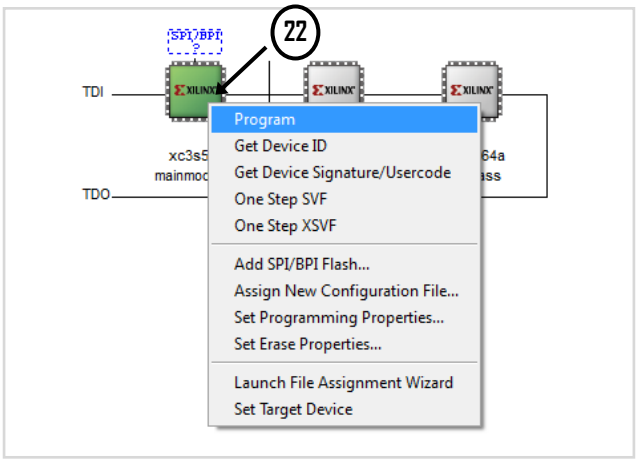


FIGURA 164 – Programar una FPGA, acció de programar.

23. Clic *OK*.

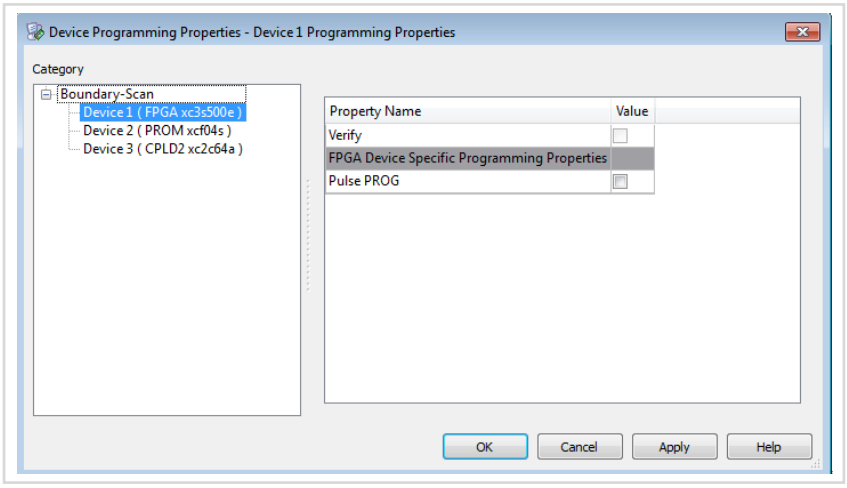


FIGURA 165 – Programar una FPGA, confirmar propietats.

24. Clic amb el botó dret sobre el primer component, i en el menú desplegable clic sobre *Set Target Device*.

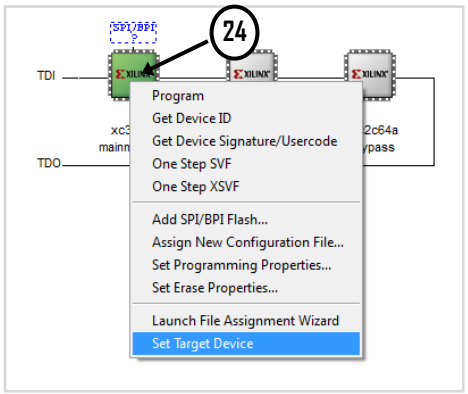


FIGURA 166 – Programar una FPGA, enviar les dades al dispositiu.

25. S'obrirà un avís com el següent. Clic *OK*

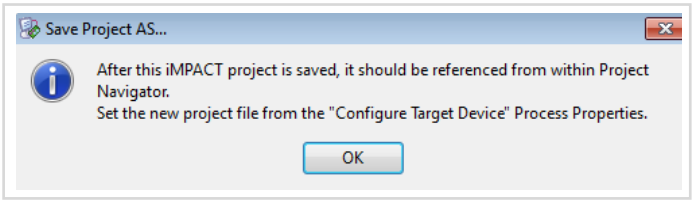


FIGURA 167 – Programar una FPGA, avís conforme caldrà aplicar les propietats al projecte.

26. S'obrirà una pantalla per a guardar els canvis realitzats. Donar-li un nom (*ex. test.ipf*) i *Guardar*.

27. Tornar a l'eina *ISE Design Suite*.
28. Clic amb el botó dret sobre *Configure Target Device*.
29. S'obrirà un menú desplegable clic sobre *Process Properties.*

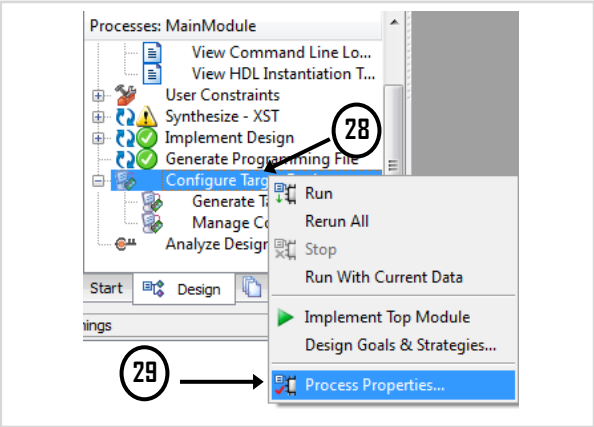


FIGURA 168 – Programar una FPGA, obrir opció d'aplicar les propietats al projecte.

30. S'obrirà una nova finestra per assignar-li en procés de configuració que hem realitzat en els passos anteriors.
31. Determinar el *iMPACT Project File*, fent clic sobre *'…'* que obrirà una nova finestra on escollir el nostre fitxer que conté el procés de configuració *(ex. test.ipf).*
32. Seleccionar el fitxer. Clic *Obre*.

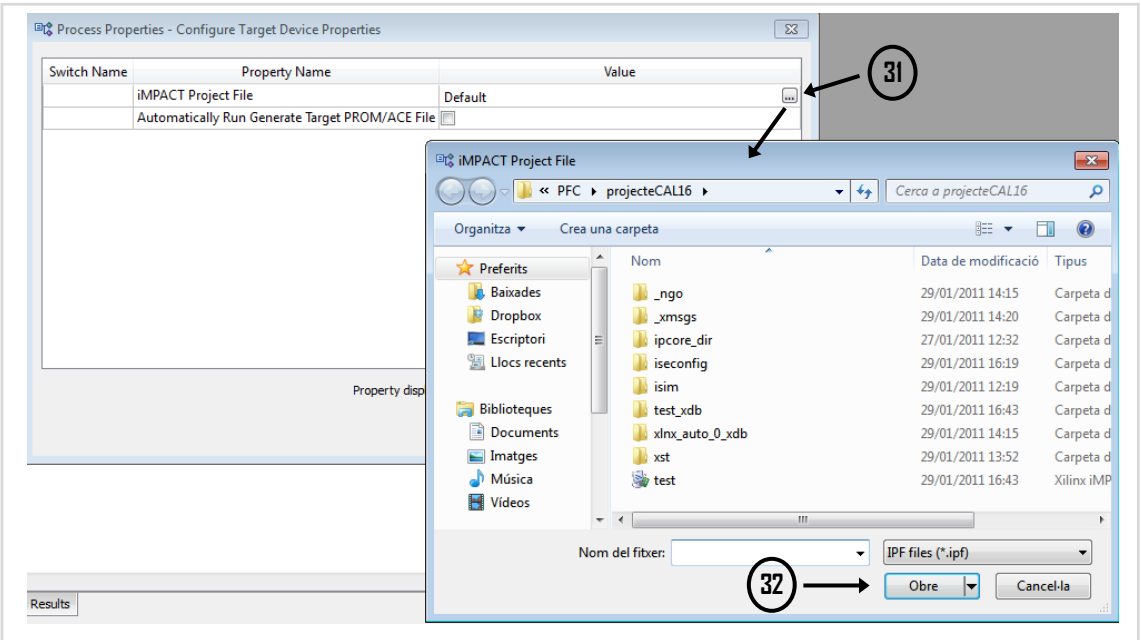


FIGURA 169 – Programar una FPGA, assignar el fitxer de propietats al projecte.

33. Doble clic a Configure Target Device.
34. El projecte ja s'ha programat a la FPGA, comprovar el seu comportament.

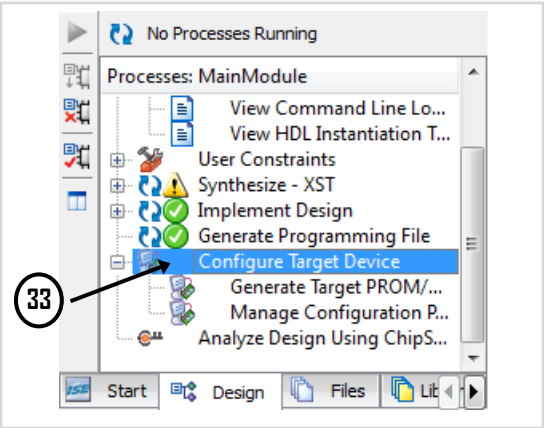


FIGURA 170 – Programar una FPGA, pas final.

TREBALLAR AMB EL PROCESSADOR CAL16

ESTRUCTURACIÓ DEL PROCESSADOR

El projecte que conté la implementació del processador CAL16, està organitzat en un conjunt de carpetes i fitxers, que contenen cadascun dels mòduls que el formen.

La jerarquia del projecte i la situació dels seus mòduls és la següent:

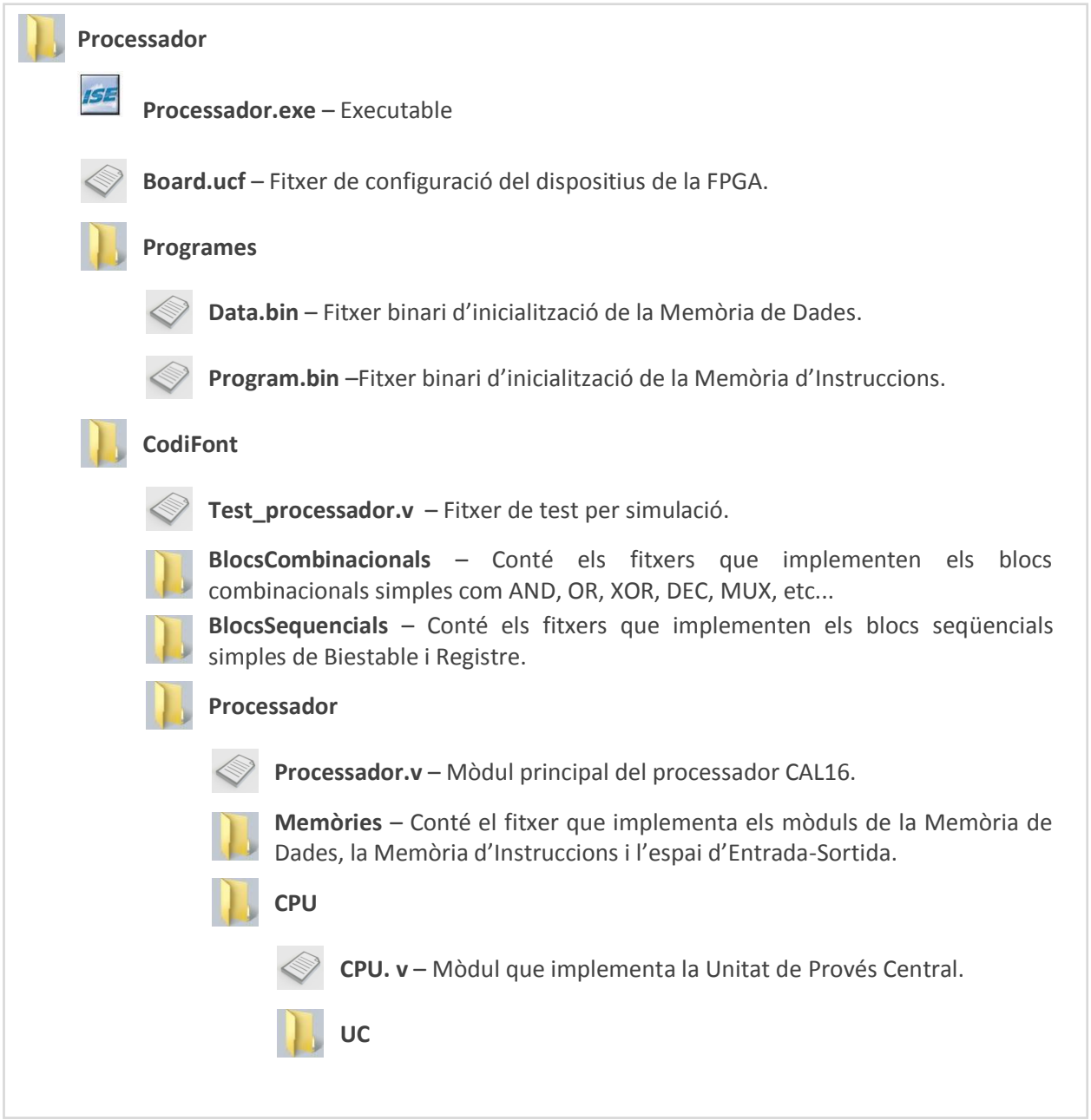


FIGURA 171 – Estructura de directoris i fitsers del processador CAL16.

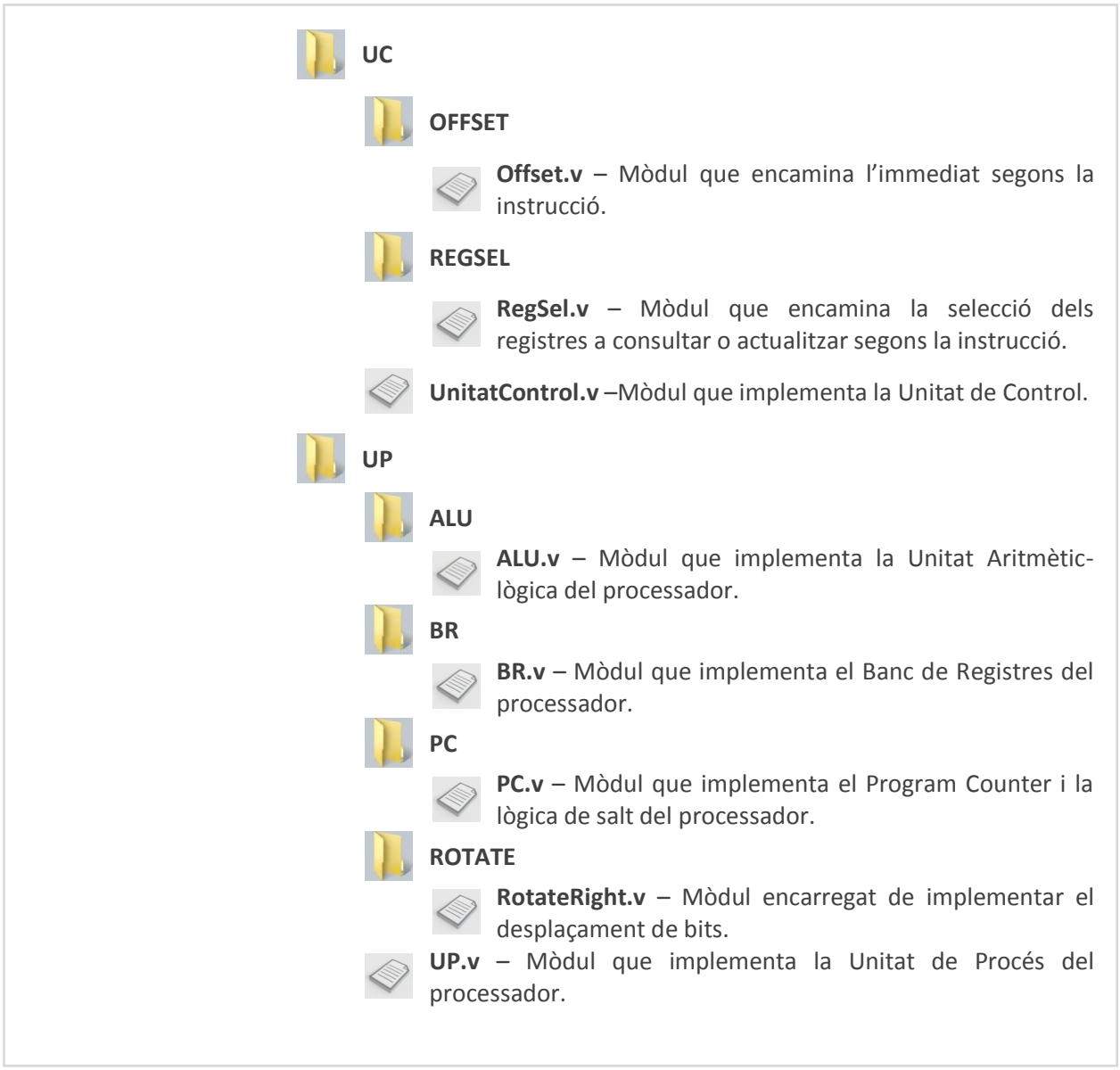


FIGURA 171 – Estructura de directoris i fitsers del processador CALI6.

PROGRAMAR EN CAL_assembler

Per a implementar un programa en llenguatge *CAL_assembler* que pugui ser interpretat pel compilador del *CAL16*, és necessari seguir els següents passos:

1. Obrir un editor, com per exemple el NotePad++.
2. Crear un document en blanc.
3. Guardar-lo :
   Directori: Compilador\jps
   Nom: <el_nom_que_sigui>
   Extensió: All types (*.*) – Important: Sense extensió .txt.
4. Programar seguint les normes de programació del *CAL_assembler*[2].

COMPILAR UN PROGRAMA IMPLEMENTAT EN CAL_assembler

Per a fer ús del compilador del *CAL16*, és necessari primer disposar del seu executable. Així doncs en cas de ja haver-ne fet ús ja anteriorment, saltar els passos 3 i 4, aquests són per a preparar el compilador, tant sols s'han de realitzar el primer cop que s'utilitzi, o bé en cas que es modifiqui el seu comportament.

1. Obrir una terminal de la consola de Windows.
   Inici -> Cerca -> cmd + Enter

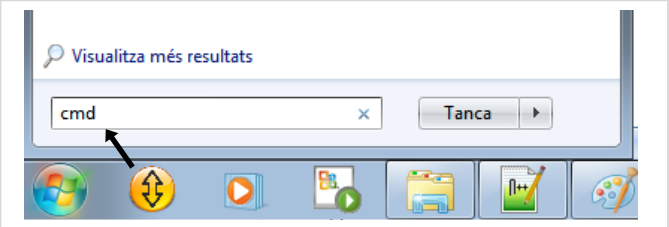


FIGURA 172 – Compilar un programa en CAL_assembler, obrir consola.

2. Col·locar-se al directori de Compilador\
   C:\> cd <ruta_processador>\Compilador

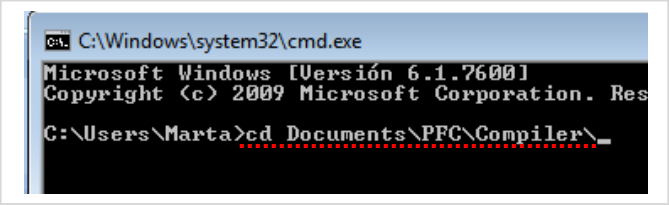


FIGURA 173 – Compilar un programa en CAL_assembler, exemple de com obrir el directori.

[2] *CAL_assembler: Per amés informació sobre el llenguatge CAL_assembler, veure capítol Memòria Tècnica, apartat CAL_assembler.*

3. A la línea de comandes executar: **C:\> make clean**
4. A la línea de comandes executar: **C:\> make all**

```
C:\Users\Marta\Documents\PFC\Compiler\CompiladorPFC>
C:\Users\Marta\Documents\PFC\Compiler\CompiladorPFC>make clean
rm -f *.o
rm -f *.exe
rm -f cl.c scan.c parser.dlg

C:\Users\Marta\Documents\PFC\Compiler\CompiladorPFC>make all
Antlr.exe -gt cl.g
Antlr parser generator   Version 1.33MR33   1989-2001
Dlg.exe -ci parser.dlg scan.c
dlg  Version 1.33MR33   1989-2001
g++ -IC:\Users\Marta\Documents\PFC\Compiler\pccts\h -IC:\Users\Marta\Documents\P
FC\Compiler\pccts -Wno-write-strings -c -o cl.o cl.c
g++ -IC:\Users\Marta\Documents\PFC\Compiler\pccts\h -IC:\Users\Marta\Documents\P
FC\Compiler\pccts -Wno-write-strings -c -o codegen.o codegen.cc
g++ -IC:\Users\Marta\Documents\PFC\Compiler\pccts\h -IC:\Users\Marta\Documents\P
FC\Compiler\pccts -Wno-write-strings -c -o ptype.o ptype.cc
g++ -IC:\Users\Marta\Documents\PFC\Compiler\pccts\h -IC:\Users\Marta\Documents\P
FC\Compiler\pccts -Wno-write-strings -c -o semantic.o semantic.cc
g++ -IC:\Users\Marta\Documents\PFC\Compiler\pccts\h -IC:\Users\Marta\Documents\P
FC\Compiler\pccts -Wno-write-strings -c -o scan.o scan.c
g++ -IC:\Users\Marta\Documents\PFC\Compiler\pccts\h -IC:\Users\Marta\Documents\P
FC\Compiler\pccts -Wno-write-strings -c -o err.o err.c
g++ -o cl cl.o codegen.o ptype.o semantic.o scan.o err.o
Información: se resuelve std::cout  al enlazar con __imp___ZSt4cout (auto-import
ación)
c:/mingw/bin/../lib/gcc/mingw32/4.5.0/../../../../mingw32/bin/ld.exe: aviso: la
importación automática se activó sin especificar --enable-auto-import en la líne
a de órdenes.
Esto debe funcionar a menos que involucre estructuras de datos constantes que re
ferencíen símbolos de DLLs auto-importadas.

C:\Users\Marta\Documents\PFC\Compiler\CompiladorPFC>_
```

FIGURA 174 – Compilar un programa en CAL_assembler, comandes del MakeFile.

5. Col·locar el programa a compilat a la carpeta Compilador\jps
   Ex: Compilador\jps\el_meu_programa
6. A la línea de comandes executar: **C:\> cl.exe < jps\el_meu_programa**

```
>cl.exe < jps\jp_cercaMax
```

FIGURA 174 – Compilar un programa en CAL_assembler, exemple de com executar la compilació d'un programa.

7. En cas que el programa tingui errors, el compilador ho indicarà d'una de les següents maneres en funció del tipus d'error i identificarà els errors a la mateixa consola. Cal corregir els errors i tornar al pas 5.

```
C:\Users\Marta\Documents\PFC\Compiler\CompiladorPFC>cl.exe < jps\jp_elMeuProgram
a


  1: li r1,128  //0
  2: lih r1,0   //1 R1 = 0x0080
  3: li r2,10   //2
  4: st 0(r1),r2        //3 leds = 10
  5: jmpi                 //4
  6: line 6: syntax error at "addi" missing { REG LABEL IMM NEG } <-- ERROR
addi r4,r4,r4
  7:
  8:
  9:
 10:
 11:
There were syntax errors.
```

FIGURA 175 – Compilar un programa en CAL_assembler, exemple d'error de compilació de tipus sintàctic.

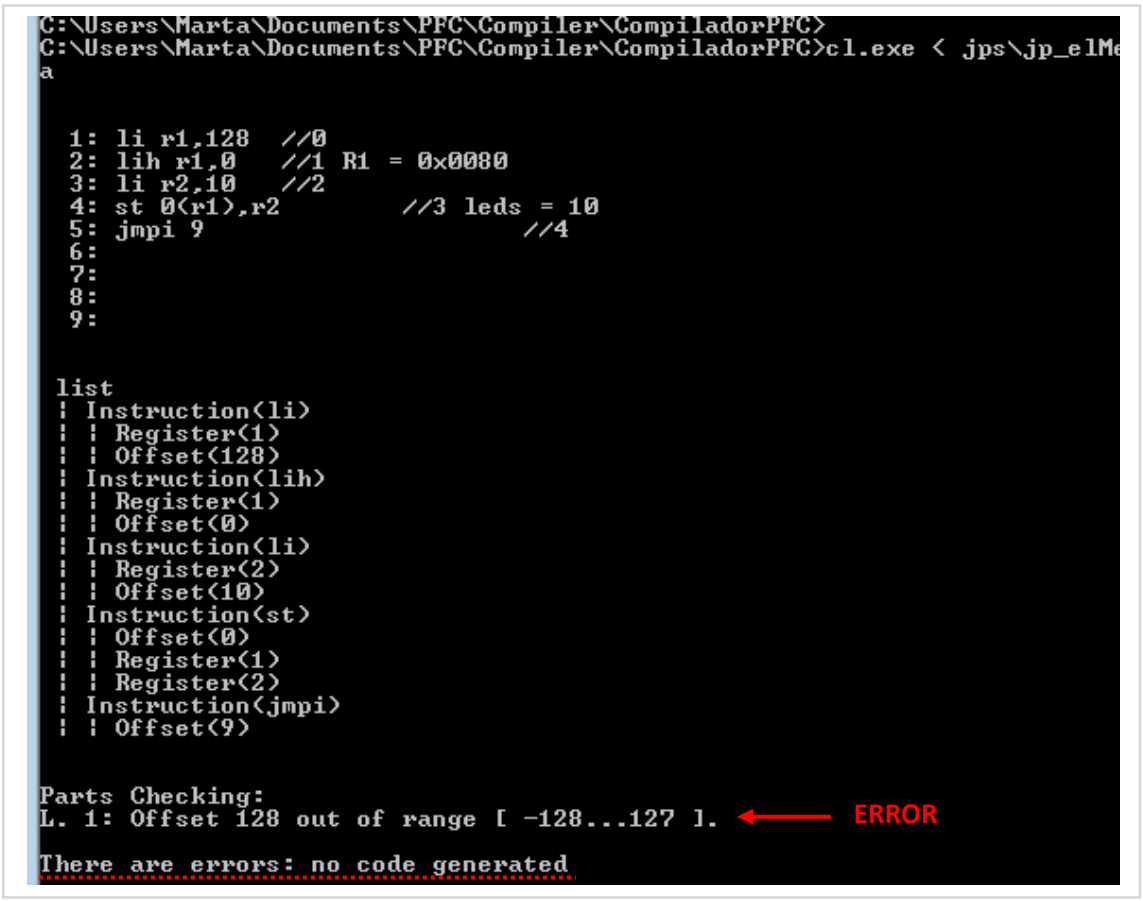


FIGURA 176 – Compilar un programa en CAL_assembler, exemple d'error de compilació de tipus semàntic.

8. En cas de no haver-hi errors el propi compilador ja haurà col·locat el fitxer binari associat el teu programa a la carpeta **Processador\programes** del processador amb el nom de **program.bin.**
   ! Si ja existeix un altre **program.bin** en aquesta carpeta l'existent serà sobreescrit per l'actual.

FIGURA 177 – Compilar un programa en CAL_assembler, exemple de compilació sense errors.

SIMULAR I PROGRAMAR UNA FPGA AMB EL PROCESSADOR CAL16

És necessari que el fitxer binari associat al programa que volem executar, és a dir, el que genera el compilador un cop el nostre programa no té errors, s'anomeni *program.bin* i es trobi a la carpeta de *Processador**Programes* del nostre processador.

Si es vol implementar el fitxer *program.bin* sense utilitzar el compilador, és cal complir:

- Aquest fitxer conté 4096 línies que inicialitzen respectivament les 4096 paraules de la Memòria d'Instruccions del processador.
- Cada línea està formada per una tira de 16 bits i codifica una instrucció seguint les normes de traducció definides pel llenguatge *CAL_assembler*.
- La primera instrucció a executar es troba a la primera línea (adreça 0x000 de la Memòria d'Instruccions)
- Per cada línea els bits estan situat de més a menys pes, mirant de dreta cap esquerra respectivament.
- Les instruccions s'executen de forma consecutiva, menys en el punt que es troba amb una instrucció de ruptura de seqüència.
- El fitxer ha de disposar de totes les posicions de memòria inicialitzades, en cas de que en una direcció concreta no s'hagi d'inicialitzar amb cap instrucció la seva inicialització serà amb 16 zeros.

Pel que fa al fitxer *data.bin* situat a la mateixa carpeta *Processador**Programes*, conté la inicialització de la Memòria de Dades. Si es vol donar algun valor a aquesta, abans d'executar el programa s'ha d'escriure en aquest fitxer el valor del contingut amb el que es vol inicialitzar. Cal tenir en compte que:

- Aquest fitxer conté 256 línies que inicialitzen respectivament les 256 paraules de la Memòria de Dades del processador.
- Cada línea està formada per una tira de 16 bits i codifica el valor que conté en binari.
- La primera línea correspon a la direcció 0 de la Memòria de Dades.
- Per cada línea els bits estan situat de més a menys pes, mirant de dreta cap esquerra respectivament.

Per a executar el comportament del processador *CAL16* en funció del nostre programa, un cop ja disposem dels dos fitxers anteriors ben inicialitzats i col·locats en el directori correcte:

1. Obrir el projecte fent clic a la icona d'execució que es troba en el directori arrel *Processador*\. 
2. Un cop obert l'eina de ISE Desgin Suit, seguir els passos definits en els apartats anteriors d'aquest mateix manual d'usuari de:
   - Executar amb ISE Simulator.
   - Programar una FPGA

COL·LECCIÓ DE PROBLEMES

ÍNDEX

PROGRAMACIÓ EN VERILOG

OBJECTIU

Aquest apartat té com a objectiu principal conèixer el funcionament de circuits combinacionals i seqüencials, saber dissenyar-los per a que es comportin de la forma especificada i aprendre a programar-los en Verilog. Alhora permet familiaritzar-se amb les eines de Xilinx ja sigui a nivell de desenvolupament com a nivell de simulació dels mòduls programats.

EXERCICIS

EXERCICI 1: MUX_2-1_4b i MUX_8-3_4b

ENUNCIAT: Crea dos nous mòduls anomenats **MUX_2-1_4b** i **MUX_8-1_4b** que implementin un multiplexor de 2 entrades i 1 sortida, i un de 8 entrades i 1 sortida, respectivament. Els busos d'entrada i de sortida dels dos blocs són de 4 bits. Pots basar-te en la implementació a presentada en aquesta memòria del **MUX_2-1_1b** per implementar-los.

MUX_2-1_1b

```
module mux2_1_1bit(a, b, sel, out);
  output out;
  input a,b,sel;

  not I5 (sel_n, sel);
  and I6 (sel_a, a, sel);
  and I7 (sel_b, b, sel_n);
  or I4 (out, sel_a, sel_b);
endmodule
```

sel	a	b	out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

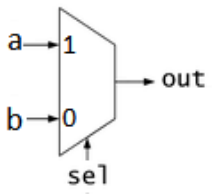


FIGURA 178 – Exercici I enunciat. Especificació bloc MUX_2-1_1b.

En paper:

a) Dibuixa els blocs MUX_2-1_4b i MUX_8-1_4b que encapsularan cadascuna de les teves implementacions. No t'oblidis d'indicar el nombre de bits dels busos d'entrada i dels de sortida.

b) Emplena la següent taula on es mostra el funcionament d'un multiplexor2-1 de 4 bits. Els busos A,B i Out són de 4 bits i les dades X i Y també.

A	B	Sel	Out
X	Y	0	
X	Y	1	

c) Fes una taula com l'anterior per mostrar el comportament del bloc MUX_8-3_4b. Pensa quantes entrades i sortides de 4 bits ha de tenir? Quants bits ha de tenir el senyal Sel?

d) Dissenya les implementacions internes dels mòduls anteriors, utilitzant els propis blocs definits i el MUX_2-1_1b.

Amb Xilinx ISE Design Suite:

e) Programa en Verilog els dissenys anteriors.

f) Crea un fitxer de test pel mòdul MUX_8-1_4b on comprovis el seu funcionament.

g) Simula i verifica el seu correcte comportament. Per analitzar més fàcilment el resultat mostra els valors del cronograma en hexadecimal.

SOLUCIÓ:

a)

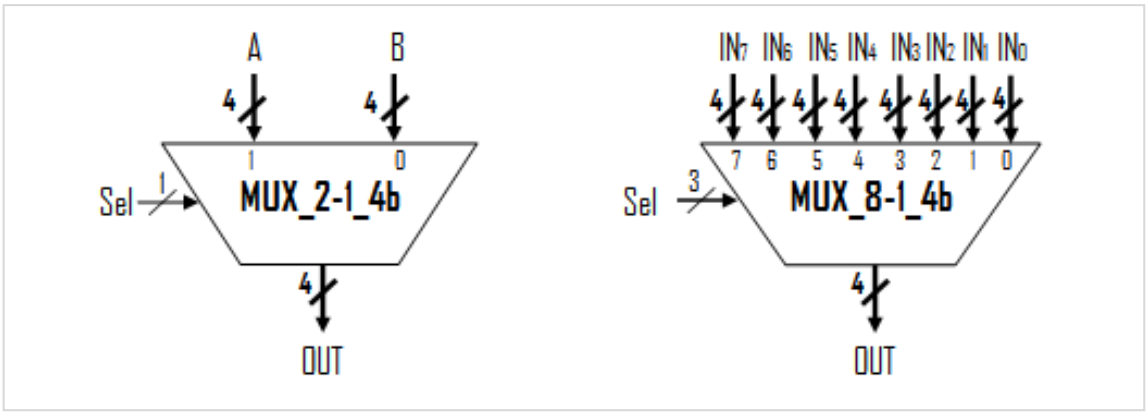


FIGURA 179 – Exercici I solució a). Bloc MUX_2-1_4b i bloc MUX_8-1_4b.

b)

A	B	Sel	Out
X	Y	0	**Y**
X	Y	1	**X**

on X i Y són valors de 4 bits

FIGURA 180 – Exercici I solució b). Taula de Veritat bloc MUX_2-1_4b.

c)

IN7	IN6	IN5	IN4	IN3	IN2	IN1	IN0	Sel0	Sel1	Sel2	OUT
A	B	C	D	E	F	G	H	0	0	0	**H**
A	B	C	D	E	F	G	H	0	0	1	**G**
A	B	C	D	E	F	G	H	0	1	0	**F**
A	B	C	D	E	F	G	H	0	1	1	**E**
A	B	C	D	E	F	G	H	1	0	0	**D**
A	B	C	D	E	F	G	H	1	0	1	**C**
A	B	C	D	E	F	G	H	1	1	0	**B**
A	B	C	D	E	F	G	H	1	1	1	**A**

*A, B, C, D, E, F, G i H són valors de 4 bits. INx on x={0..7} són entrades de 4 bits i OUT és una sortida de 4 bits.

FIGURA 181 – Exercici I solució c). Taula de Veritat bloc MUX_8-1_4b.

d)

MUX_2-1_4b implementat a partir de **MUX_2-1_1b**

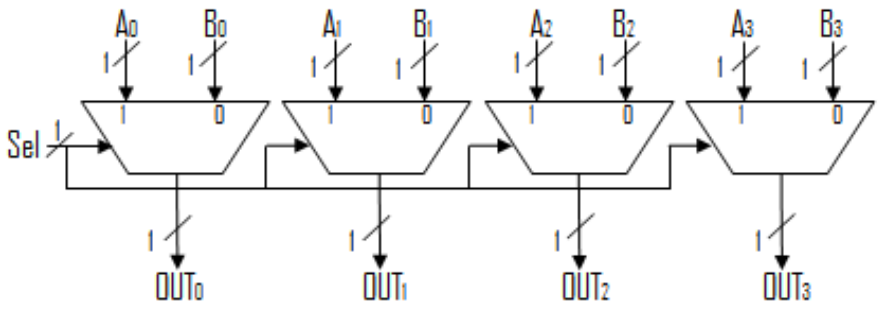


FIGURA 182 – Exercici I solució d). Implementació interna del bloc MUX_2-1_4b.

MUX_8-1_4b implementat a partir de **MUX_2-1_4b**

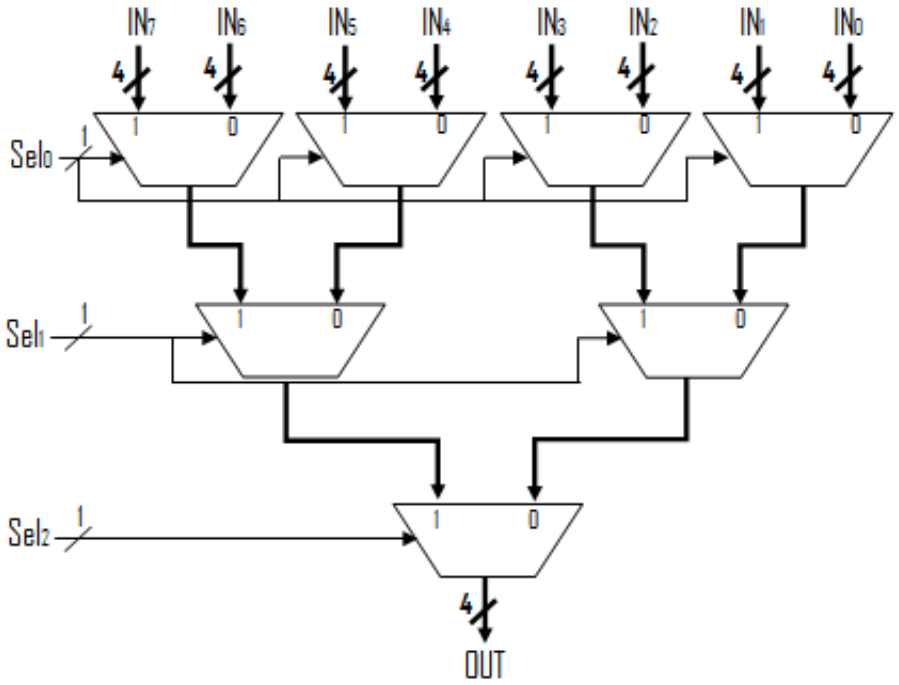


FIGURA 183 – Exercici I solució d). Implementació interna del bloc MUX_8-1_4b.

e)

```
module mux2_1_4bit(a, b, sel, out);
    output [3:0] out;
    input [3:0] a,b;
    input sel;

    mux2_1_1bit bit0 (a[0], b[0], sel, out[0]);
    mux2_1_1bit bit1 (a[1], b[1], sel, out[1]);
    mux2_1_1bit bit2 (a[2], b[2], sel, out[2]);
    mux2_1_1bit bit3 (a[3], b[3], sel, out[3]);
endmodule
```

FIGURA 184 – Exercici I solució e). Implementació en Verilog del bloc MUX_2-1_4b.

```
module mux8_1_4bit( in0, in1, in2, in3, in4, in5, in6, in7, sel, out);
    output [3:0] out;
    input [3:0] in0,in1,in2,in3,in4,in5,in6,in7;
    input [2:0] sel;

    wire [3:0] out0,out1,out2,out3,out4,out5;

    mux2_1_4bit mux0 (in1, in0, sel[0], out0);
    mux2_1_4bit mux1 (in3, in2, sel[0], out1);
    mux2_1_4bit mux2 (in5, in4, sel[0], out2);
    mux2_1_4bit mux3 (in7, in6, sel[0], out3);

    mux2_1_4bit mux4 (out1, out0, sel[1], out4);
    mux2_1_4bit mux5 (out3, out2, sel[1], out5);

    mux2_1_4bit mux6 (out5, out4, sel[2], out);
endmodule
```

FIGURA 185 – Exercici I solució e). Implementació en Verilog del bloc MUX_8-1_4b.

f)

```
module TestModule;
    reg [3:0] IN0,IN1,IN2,IN3,IN4,IN5,IN6,IN7;
    reg [2:0] sel;
    wire [3:0] OUT;

    mux8_1_4bit MUX (IN0,IN1,IN2,IN3,IN4,IN5,IN6,IN7,sel,OUT);

    initial begin
        IN0 = 4'h8;
        IN1 = 4'h9;
        IN2 = 4'hA;
        IN3 = 4'hB;
        IN4 = 4'hC;
        IN5 = 4'hD;
        IN6 = 4'hE;
        IN7 = 4'hF;
        sel = 3'd0;

        #5 sel = 3'd1;
        #5 sel = 3'd2;
        #5 sel = 3'd3;
        #5 sel = 3'd4;
        #5 sel = 3'd5;
        #5 sel = 3'd6;
        #5 sel = 3'd7;
    end
endmodule
```

FIGURA 186 – Exercici I solució f). Implementació en Verilog de fitxer de test pel bloc MUX_8-1_4b.

g)

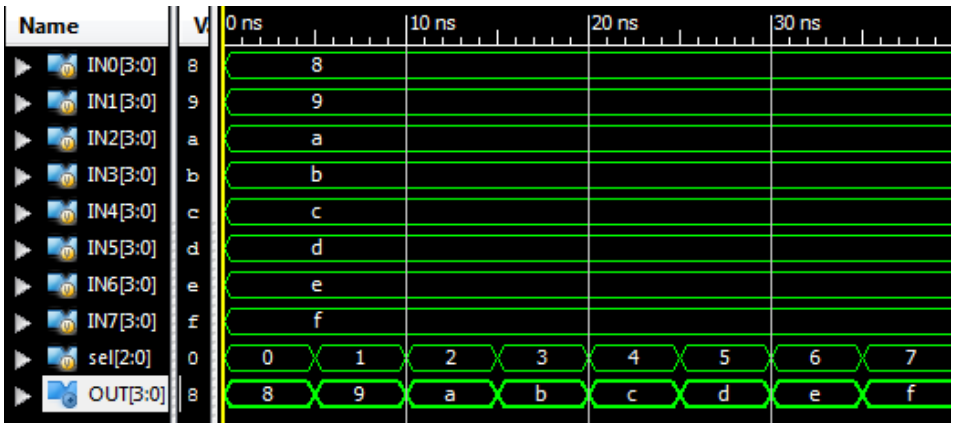


FIGURA 187 – Exercici I solució g). Cronograma resultant de l'execució del fitxer de test pel bloc MUX_8-1_4b.

EXERCICI 2: Mòdul SUB

ENUNCIAT: Crea un nou mòdul anomenat **SUB** encarregat de realitzar la resta entre dos nombres enters codificats en Ca2. Els 2 busos d'entrada són de 4 bits i cal donar el resultat en un bus de sortida de 4 bits. A més el bloc disposa de 2 senyals més de sortida, un que indica si la operació de resta genera borrow (b) i un que indica si hi ha desbordament en la representació del resultat (ovf), tal i com presenta la següent especificació:

Entrada:	Bus **A** [4 bits]	
	Bus **B** [4 bits]	
Sortida:	Bus **W** [4 bits]	–> W = B - A
	Senyal **b** [1 bit]	–> b = (B – A) genera borrow?
	Senyal **ovf** [1 bit]	–> ovf = (B – A) no és representable en 4 bits?

Concretament segueix els següents passos:

En un paper:

a) Dibuixa el bloc SUB que encapsularà la teva implementació. No t'oblidis d'indicar el nombre de bits dels busos d'entrada i dels de sortida.

b) Dissenya la seva implementació interna.

Amb Xilinx ISE Design Suite:

c) Programa en Verilog el disseny anterior.

d) Crea un fitxer de test pel teu mòdul on comprovis totes les possibles combinacions de signes dels operands d'entrada i els casos extrems que consideris necessaris.

e) Simula i verifica el seu correcte comportament. Per analitzar més fàcilment el resultat mostra els valors del cronograma en decimal amb signe.

SOLUCIÓ:

a)

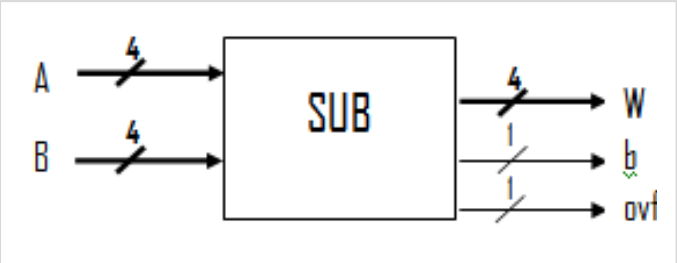


FIGURA 188 – Exercici 2 solució a). Bloc SUB.

b) Una possible solució pot ser utilitzar el bloc Full-Adder presentat en el capítol d'Introducció al Verilog de la Memòria Tècnica.

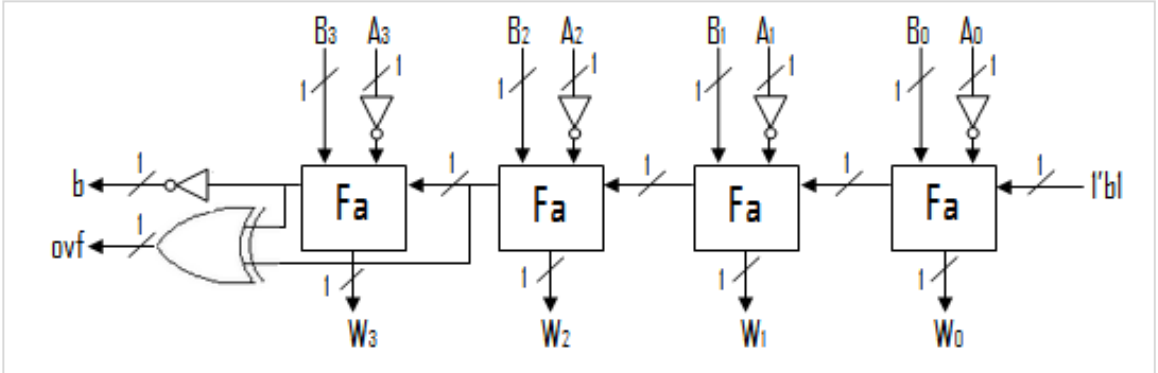


FIGURA 189 – Exercici 2 solució b). Implementació interna del bloc SUB_4b.

c)

```
module SUB_4b(A,B,W,b,ovf);
    input [3:0] A,B;
    output [3:0] W;
    output b,ovf;

    wire b0,b1,b2,b3;
    wire [3:0] A_n;

    not_1bit not0(A[0],A_n[0]);
    not_1bit not1(A[1],A_n[1]);
    not_1bit not2(A[2],A_n[2]);
    not_1bit not3(A[3],A_n[3]);

    fullAdder bit0 (A_n[0],B[0],1'b1,W[0],b0);
    fullAdder bit1 (A_n[1],B[1],b0,W[1],b1);
    fullAdder bit2 (A_n[2],B[2],b1,W[2],b2);
    fullAdder bit3 (A_n[3],B[3],b2,W[3],b3);

    not_1bit not4(b3,b);
    xor_1b xor_ovf (b3,b2,ovf);
endmodule
```

FIGURA 190 – Exercici 2 solució c). Implementació en Verilog del bloc SUB_4b.

d)

```
module TestModule;
    reg [3:0] A,B;
    wire [3:0] W;
    wire b,ovf;

    SUB_4b RESTADOR (A,B,W,b,ovf);

    initial begin
        A = 4'b0100;
        B = 4'b0010;

        #5 A = 4'b1100;
        B = 4'b1011;

        #5 A = 4'b1011;
        B = 4'b0110;

        #5 A = 4'b1111;
        B = 4'b0110;

        #5 A = 4'b1111;
        B = 4'b0111;
    end
endmodule
```

FIGURA 191 – Exercici 2 solució d). Implementació en Verilog del fitxer de test pel bloc SUB_4b.

e)

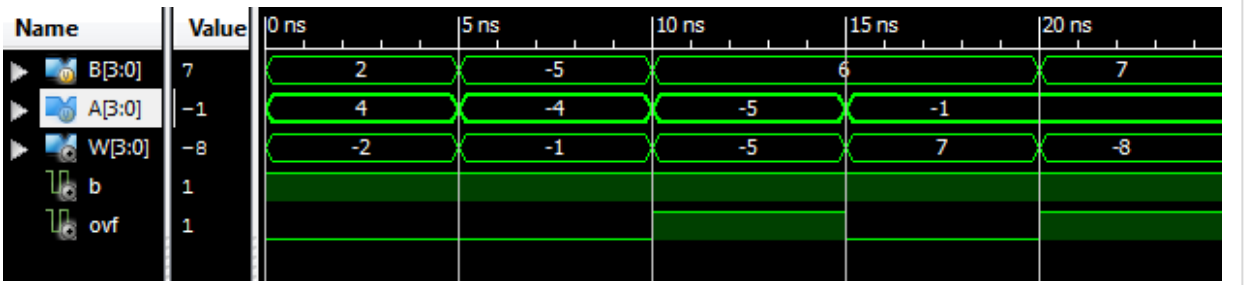


FIGURA 192 – Exercici 2 solució e). Cronograma resultant de l'execució del fitxer de test pel bloc SUB_4b.

EXERCICI 3: ÉS DIVISIBLE ENTRE 3?

ENUNCIAT: Dissenya un circuit que donat una seqüència de bits d'entrada, activi una senyal que ens indiqui que el nombre codificat per la seqüència que ha arribat fins al moment és divisible o no entre 3, concretament aquesta senyal s'anomena **d** i funciona:

d = 1 -> El nombre format per la seqüència d'entrada és divisible entre 3.
d = 0 -> El nombre format per la seqüència d'entrada NO és divisible entre 3.

Pel que fa a la seqüència d'entrada, a cada flanc de rellotge ens arriba un nou bit (1 o 0) que compon el nombre total. El nombre total es tracta com un nombre natural codificat en binari. Cada nou bit que arriba passa a ser el de menor pes del nombre que es porta acumulat. No cal emmagatzemar el nombre total, sinó tant sols canviar d'estat en funció del bit que entra. El següent exemple mostra el comportament:

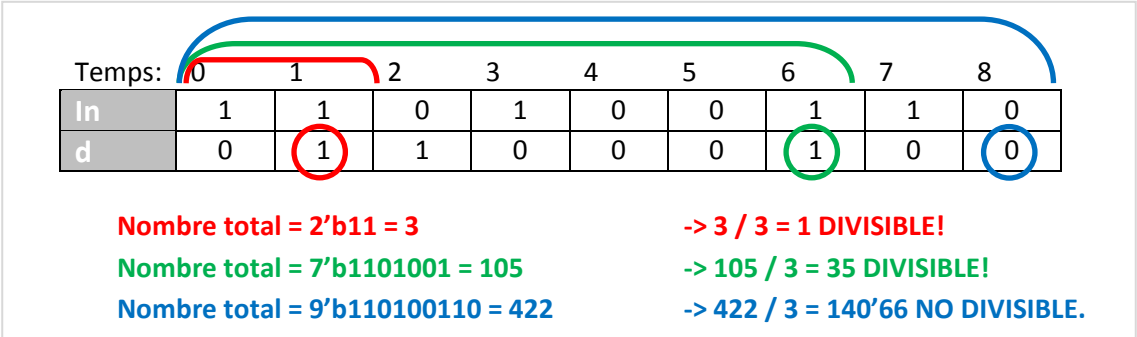


FIGURA 193 – Exercici 3 enunciat. Cronograma exemple de funcionament del bloc DIV3.

PISTA: Al dividir un nombre entre tres, obtenim un màxim de 3 residus, residu = 0 que significa que és divisible i per tant d=1, o bé residu = 1 o residu = 2 quan no s'han d'activar d.

En un paper:

a) Dibuixa el bloc **DIV3** que encapsularà la teva implementació. No t'oblidis d'indicar el nombre de bits dels busos d'entrada i dels de sortida.
b) Dibuixa el graf d'estats que guia aquest procés, ha d'incloure la llegenda.
c) Dissenya la implementació interna del teu bloc.

Amb Xilinx ISE Design Suite:

d) Programa en Verilog el disseny anterior. Pots utilitzar el biestable presentat en aquesta memòria.
e) Crea un fitxer de test pel teu mòdul on comprovis totes les possibles combinacions de signes dels operands d'entrada i els casos extrems que consideris necessaris.
f) Simula i verifica el seu correcte comportament..

SOLUCIÓ:

a)

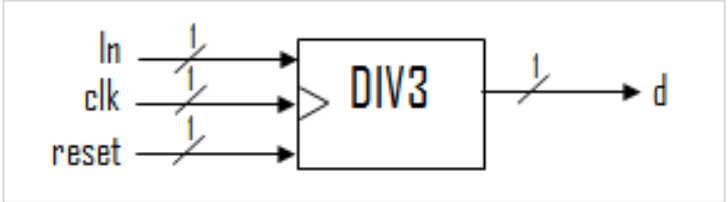


FIGURA 194 – Exercici 3 solució a). Bloc DIV3.

b)

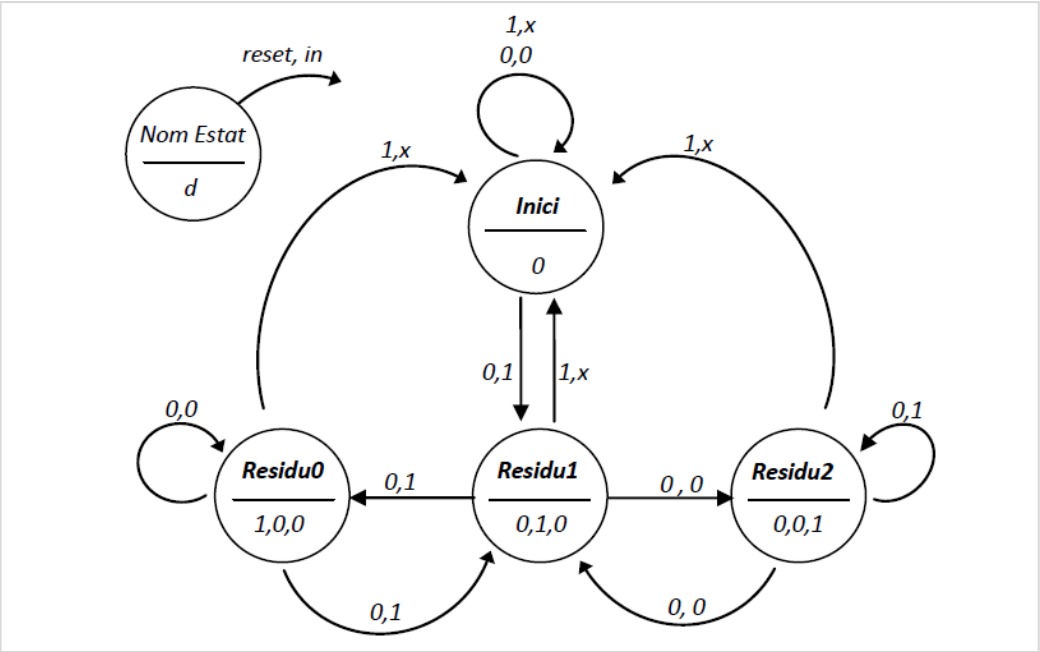


FIGURA 195 – Exercici 3 solució b). Graf d'estats corresponen al comportament del bloc DIV3.

c)

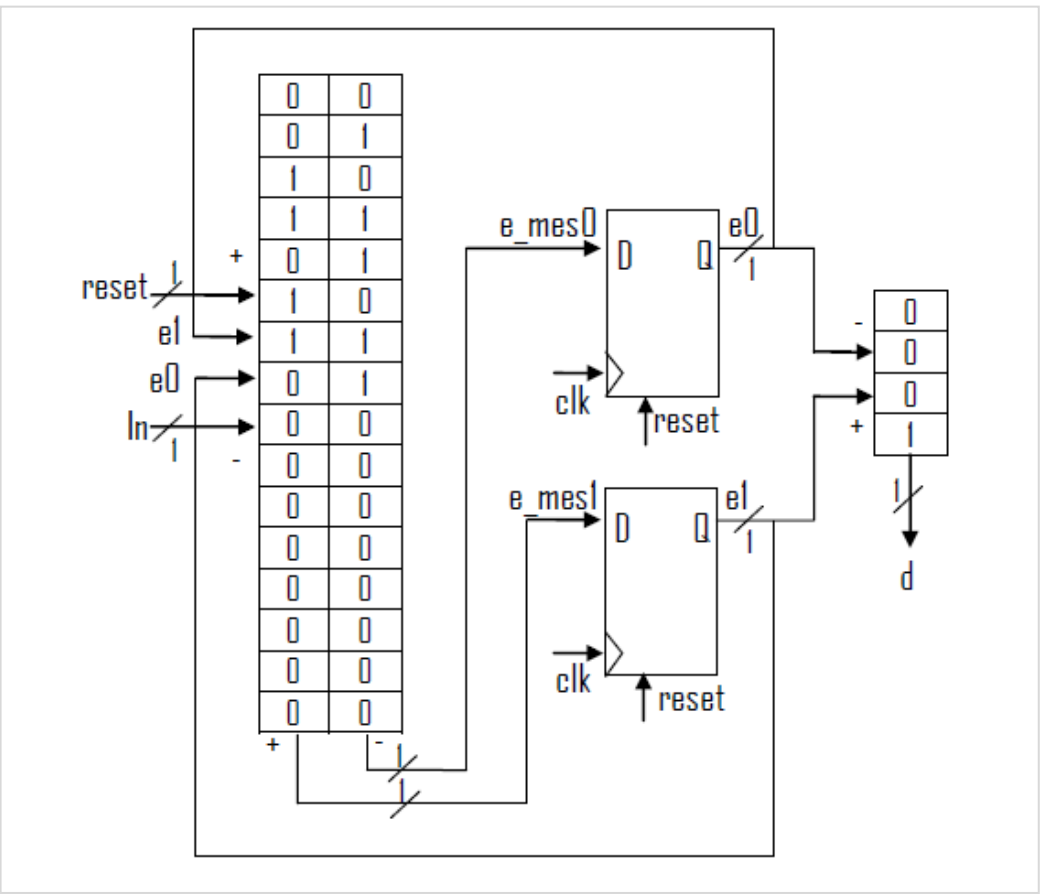


FIGURA 196 – Exercici 3 solució c). Implementació interna del bloc DIV3.

d)

```
module DIV3(in, clk, reset, d);
    input in, clk, reset;
    output d;

    wire e0,e_mes0,e1,e_mes1;
    reg [1:0] estat_seg [15:0];
    reg sortida [3:0];

    initial begin
     sortida[0] <= 1'b0;
     sortida[1] <= 1'b0;
     sortida[2] <= 1'b0;
     sortida[3] <= 1'b1;

     estat_seg[0] <= 2'b00;
     estat_seg[1] <= 2'b01;
     estat_seg[2] <= 2'b10;
     estat_seg[3] <= 2'b11;
     estat_seg[4] <= 2'b01;
     estat_seg[5] <= 2'b10;
     estat_seg[6] <= 2'b11;
     estat_seg[7] <= 2'b01;
     estat_seg[8] <= 2'b00;
     estat_seg[9] <= 2'b00;
     estat_seg[10] <= 2'b00;
     estat_seg[11] <= 2'b00;
     estat_seg[12] <= 2'b00;
     estat_seg[13] <= 2'b00;
     estat_seg[14] <= 2'b00;
     estat_seg[15] <= 2'b00;
    end

    assign e_mes0 = estat_seg[{reset,e1,e0,in}][0];
    assign e_mes1 = estat_seg[{reset,e1,e0,in}][1];
    assign d = sortida[{e1,e0}];

    ff ff0 (e_mes0,clk,reset,e0);
    ff ff1 (e_mes1,clk,reset,e1);
endmodule
```

FIGURA 197 – Exercici 3 solució d). Implementació en Verilog del bloc DIV3.

e)

```
module TestModule;
    reg clk,in,reset;
    wire d;

    DIV3 inst_DIV3 (in, clk, reset, d);

    initial begin
        reset = 1'b1;
        in = 1'b1;
        #3 reset = 1'b0;

        #2 in = 1'b1;
        #2 in = 1'b0;
        #2 in = 1'b1;
        #2 in = 1'b0;
        #2 in = 1'b0;
        #2 in = 1'b1;
        #2 in = 1'b1;
        #2 in = 1'b0;
    end

    initial clk <= 1;
    always begin
        #1 clk <= ~clk;
    end
endmodule
```

FIGURA 198 – Exercici 3 solució e). Implementació en Verilog del fitxer de test pel bloc DIV3.

f)

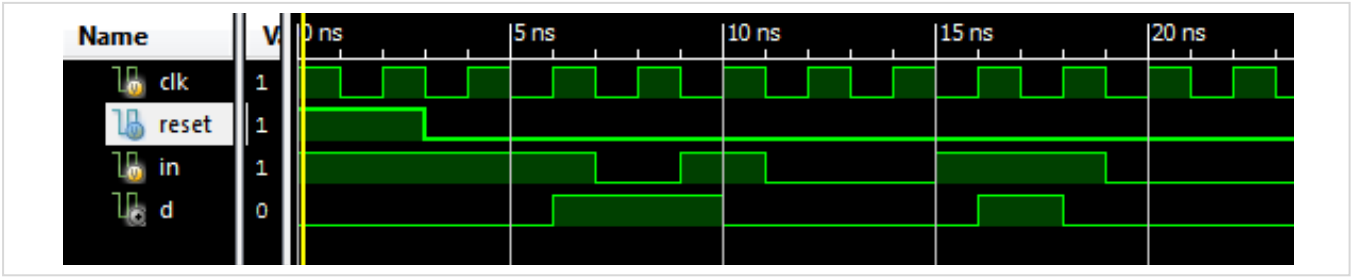


FIGURA 199 – Exercici 3 solució f). Cronograma resultant de l'execució del fitxer de test pel bloc DIV3.

SIMULACIÓ DE PROGRAMES EN CAL_assembler

OBJECTIU

Aquest apartat té com a objectiu principal aprendre a programar en *CAL_assembler* i familiaritzar-se amb el processador *CAL16*. En els següents exercicis es farà ús de les eines de simulació de Xilinx per comprovar el correcte funcionament al executar programes escrits en *CAL_assembler.*

EXERCICIS

EXERCICI 4: BUBBLE SORT

ENUNCIAT: Donat un vector de naturals amb 50 posicions. que es troba en la direcció 0 de la memòria de dades. Implementa l'algorisme d'ordenació de la bombolla, per ordenar els seus elements de petit a gran en el mateix espai, és a dir, de la posició 0 a la 49 de la Memòria de Dades. A continuació disposem del codi en C del algorisme:

```
unsigned int vec [5];

void ordenacióBombolla ()

    unsigned int aux;
    for( i=1; i<5; i++){
        for( j=0; j<5-i; j++){
        if( vec[j] > vec[j+1]){
            aux = vec[j];
            vec[j] = vec[j+1];
            vec[j+1] = aux;
        }
        }
    }
}
```

FIGURA 200 – Exercici 4 enunciat. Codi en C del programa BubbleSort.

Més concretament:

a) Tradueix a llenguatge *CAL_assembler* el codi en C que implementa l'algorisme.

b) Compila el teu programa i corregeix els errors fins que generi el fitxer binari **Program.bin.**

c) Modifica la teva Memòria de Dades amb els següents 5 valors, col·loca'ls a les 5 primeres posicions de memòria.

0x000: vec [5] = {12,2,8,3,4}

d) Executa el teu programa amb el simulador de Xilinx, primer cal que creïs un fitxer de test que generi la senyal de rellotge i controli el reset. Un cop executat, comprova que el vector resultant emmagatzemat a les posicions [0:4] de la Memòria de Dades queda ordenat de menor a major.

e) Ara modifica el teu programa perquè permeti ordenar vectors de 50 elements.

f) Actualitza el contingut de la Memòria de Dades amb un nou fitxer que et facilitem. Veure *Recursos per Exercicis – Exercici4 – f)*

g) Simula el seu comportament i comprova que el resultat en decimal és igual al de les solucions.

SOLUCIÓ:

a) Una possible traducció pot ser:

```
etiq_bombolla:
        //aux - R0
        //i   - R1
        //j   - R2
        //&vec- R3
        LI R1,1     // I = 1
        LI R3,0     // R3 = &vec
        LI R4,-5    // R4 total nombres
etiq_for1:
        ADD R5,R1,R4        // i < 5 ?
        BZ R5, etiq_fifor1
        LI R2,0             // j = 0
etiq_for2:
        ADD R5,R4,R1        // -5 + i
        ADD R5,R2,R5        // j < ( 5 - i ) ?
        BZ R5, etiq_fifor2
etiq_if:
        ADD R5,R3,R2        // R5 = &vec[j]
        LD R6,0(R5)         // R6 = vec[j]
        LD R7,1(R5)         // R7 = vec[j+1]
        LI R0,1             //x=1
etiq_do:
        ROTR R0,R0,15       //x=x>>1
        AND R8,R6,R0        //R8=A[x]
        AND R9,R7,R0        //R9=B[x]
        LI R10,-1
        XOR R10,R10,R9
        ADDI R10,R10,1      //R10=-R9
        ADD R10,R10,R8      //R10=A[x]-B[x]
        BZ R10, etiq_while  //si son iguals volta
        BZ R8, etiq_fiif    //si A[x]==0 canvia
        JMPI etiq_if        //si A[x]==1 no canvia
etiq_while:
        ADDI R10,R0,-1      //x != 1 ? volta
        BZ R10, etiq_fiif
        JMPI etiq_do
etiq_if:
        LD R0,0(R5)         // aux = vec[j]
        LD R6,1(R5)
        ST 0(R5),R6         // vec[j] = vec[j+1]
        ST 1(R5),R0         // vec[j+1] = aux
etiq_fiif:
        ADDI R2,R2,1        // j++
        JMPI etiq_for2
etiq_fifor2:
        ADDI R1,R1,1        // i++
        JMPI etiq_for1
etiq_fifor1:
        JMPI etiq_fifor1
```

FIGURA 201 – Exercici 4 solució a). Codi en CAL_assembler del programa BubbleSort.

b) Per aconseguir **program.bin** veure apartat: *Recursos per Exercicis – Exercici4 – b)*

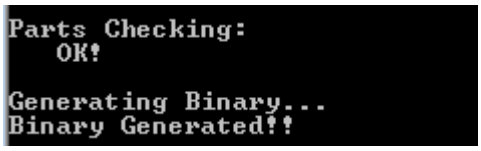


FIGURA 202 – Exercici 4 solució b). Compilació correcte del programa BubbleSort.

c) Modificar les 5 primeres línies del fitxer **Processador\Programes\data.bin** per les 5 línies del quadre.

data.bin:

```
0000000000001100
0000000000000010
0000000000001000
0000000000000011
0000000000000100
```

FIGURA 203 – Exercici 4 solució c). Primeres línies del fitxer data.bin pel programa BubbleSort.

d) Implementació del fitxer de test:

```
module test_processador();
    reg clk,reset;
    reg [3:0] sw;
    wire [7:0] led;

    Processador Cal16 (clk,sw,reset,led);

    initial begin
        reset <= 1'b1;
        #3 reset <= 1'b0;
    end

    initial clk <= 1;
    always begin
        #1 clk <= ~clk;
    end
endmodule
```

FIGURA 204 – Exercici 4 solució d). Fitxer de test en Verilog per la simulació del programa BubbleSort.

Cronograma d'execució:

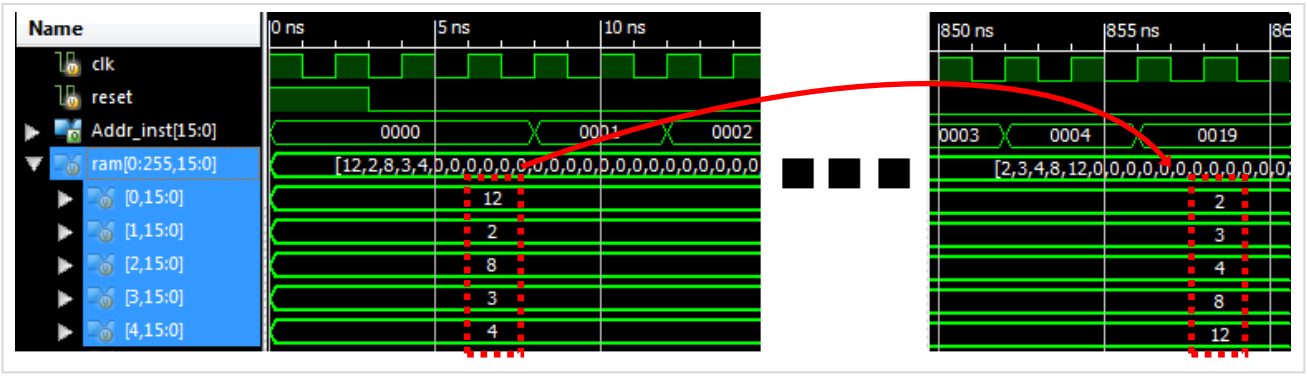


FIGURA 205 – Exercici 4 solució d). Cronograma que mostra l'execució del fitxer de test pel programa BubbleSort.

e) En el cas del programa presentat en l'aparta a, tant sols cal canviar del codi en *CAL_assembler*, el -5 per -50.

f) Modificar el contingut del fitxer **Processador\Programes\data.bin** pel contingut del nou **data.bin**. Veure *Recursos per Exercicis – Exercici4 – f)*

g)

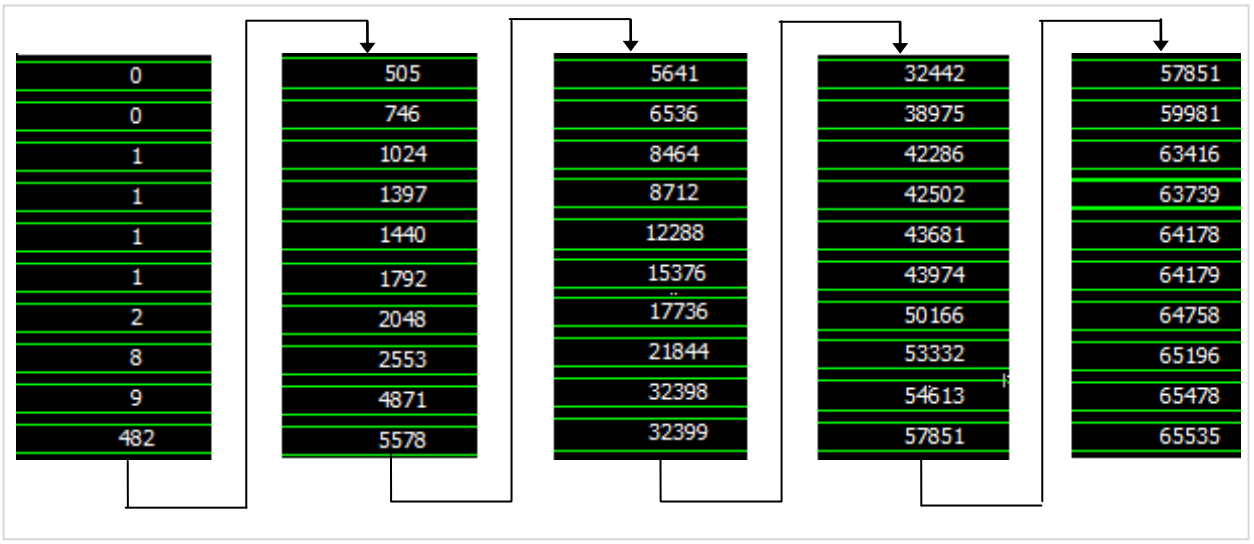


FIGURA 206 – Exercici 4 solució g). Cronograma que mostra el resultat de l'execució del fitxer de test pel programa BubbleSort.

EXERCICI 5: FUNCIÓ MULTIPLICAR

ENUNCIAT: Programa una funció que donats dos nombre emmagatzemats en les posicions 0x001 i 0x002 de la Memòria de Dades, els multipliqui i deixi el resultat a la posició 0x000 de la Memòria de Dades. A continuació disposem del codi en C del algorisme:

```
void multiplicació(){
    MemData[0] = MemData[1]*MemData[2];
}
```

FIGURA 207 – Exercici 5 enunciat. Codi en C del programa Multiplicació.

A partir d'aquí, fes els següents apartats:

a) Tradueix a llenguatge *CAL_assembler* el codi en C que implementa l'algorisme.

b) Compila el teu programa i corregeix els errors fins que generi el fitxer binari **Program.bin.**

c) Modifica la teva Memòria de Dades amb els següents valors:

MemData[1] = 10
MemData[2] = 8

d) Simula el comportament del programa, i genera el cronograma on es mostrin els valors dels registres que utilitza i el de les posicions 0, 1 i 2 de la Memòria de Dades. Quan val MemData[0] al final de l'execució?

e) Modifica el teu codi en *CAL_assembler*, per que aquest es comporti com una funció auxiliar, la qual li arriben els nombres a multiplicar per R1 i R2, i deixa el resultat de la multiplicació a R0. Implementa també en *CAL_assembler* el següent programa principal que crida a la funció multiplicació.

```
int multiplicacio ( int a, int b) { //R1=a, R2=b
    return a*b;     //Retorn per R0
}

void main(){
    leds = multiplicació(3,5);
}
```

FIGURA 208 – Exercici 5 enunciat. Codi en C de la funció auxiliar Multiplicació.

f) Compila el programa anterior, si tens errors corregeix-los fins que se't generi el **program.bin.**

g) Simula el seu comportament i genera un cronograma que contingui el valor del PC, per analitzar la crida a la subrutina, els registres que hagis utilitzat i comprova que el resultat de la multiplicació es mostra pels leds.

SOLUCIÓ:

a) Una possible traducció pot ser:

```
        LI R0,0
        LD R1,1(R0)         // R1 = MemData[1]
        LD R2,2(R0)         // R2 = MemData[2]
etiq_while:
        BZ R2, etiq_fiwhile // R2 == 0 ?
        ADD R0,R0,R1        // suma += R1
        ADDI R2,R2,-1
        JMPI etiq_while
etiq_fiwhile:
        ST 0(R2),R0         // MemData[0] = suma
etiq_fi:
        JMPI etiq_fi
```

FIGURA 209 – Exercici 5 solució a). Codi en CAL_assembler del programa Multiplicació.

b) Per aconseguir **program.bin** veure apartat: *Recursos per Exercicis – Exercici5 – b)*

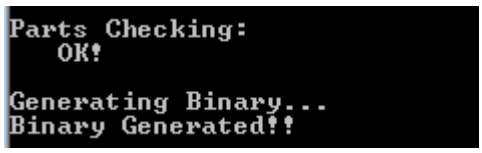


FIGURA 210 – Exercici 5 solució b). Compilació correcte del programa Multiplicació.

c) Modificar les línies del fitxer **Processador\Programes\data.bin** per les del quadre.

MemData[0] = 80;

data.bin:

```
0000000000000000
0000000000001010
0000000000001000
```

FIGURA 211 – Exercici 5 solució c). Primeres línies del fitxer data.bin pel programa Multiplicació.

d) El següent cronograma mostra el comportament de la simulació:

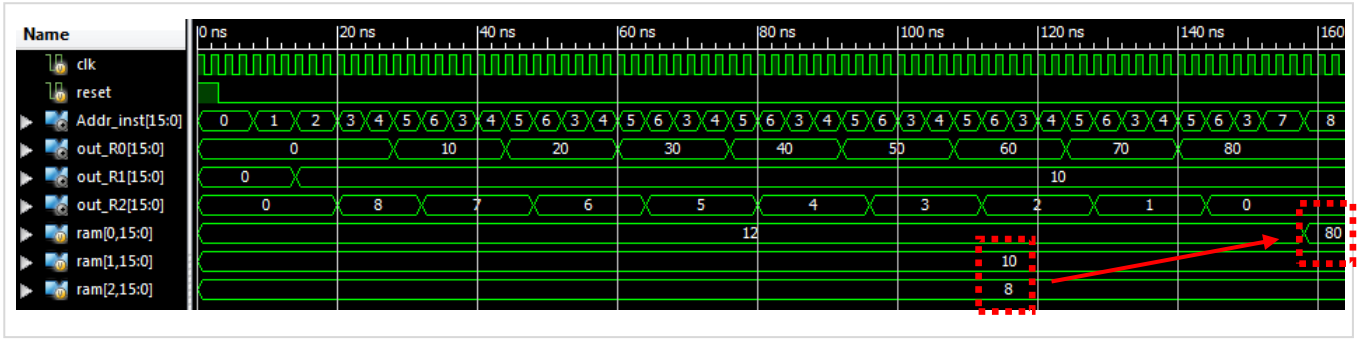


FIGURA 212 – Exercici 5 solució d).. Cronograma que mostra el resultat de l'execució del programa Multiplicació.

e)

```
etiq_main:
        LI R1,3             //Pas paràmetre R1
        LI R2,5             //Pas paràmetre R2
        LI R15,etiq_multiplicacio
        LIH R15,etiq_multiplicacio
        JR R15,R15,0        // Crida la funció i link retorn en R15
        LI R1,0
        LIH R1,1            // R1 = 0x0100 = &leds
        ST 0(R1),R0
etiq_fi:
        JMPI etiq_fi

etiq_multiplicacio:         //paràmetres en R1 i R2
        LI R0,0             // suma = 0
etiq_while:
        BZ R2, etiq_fiwhile // R2 == 0 ?
        ADD R0,R0,R1        // suma += R1
        ADDI R2,R2,-1
        JMPI etiq_while
etiq_fiwhile:
        JMP R15             //Retorna a la direcció linkada en R15
```

FIGURA 213 – Exercici 5 solució e).. Codi en CAL_assembler de la funció auxiliar Multiplicació.

f) Per aconseguir **program.bin** veure apartat: *Recursos per Exercicis – Exercici5 – f)*

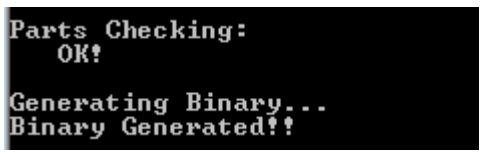


FIGURA 214 – Exercici 5 solució f). Compilació correcte de la funció auxiliar Multiplicació.

g)

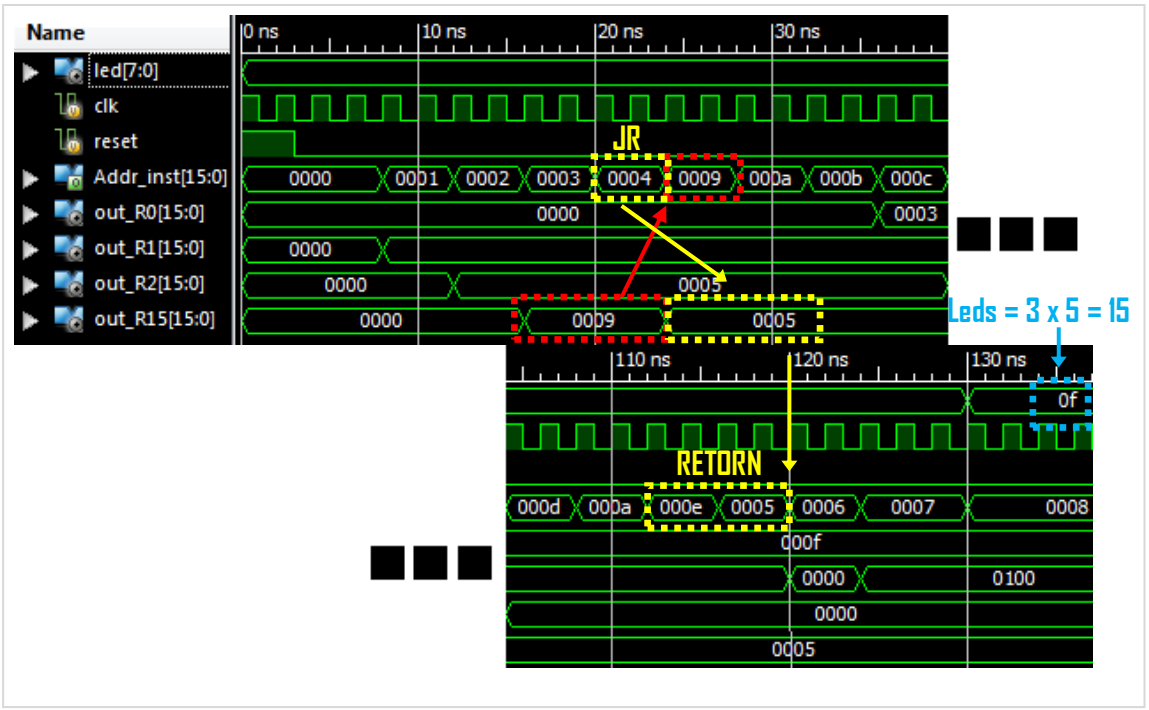


FIGURA 215 – Exercici 5 solució g). Cronograma que mostra el resultat de l'execució de la funció auxiliar Multiplicació.

EXERCICI 6: FACTORIAL D'UN NOMBRE

ENUNCIAT: Programa una funció recursiva que donat un nombre codificat amb els interruptors de menys pes (sw[2:0]), calculi el seu factorial i mostri el resultat pels leds.

Més concretament s'utilitzarà l'interruptor de més pes per indicar que el nombre està codificat i estable per ser llegit. Mentre sw[3] sigui 0, esperarem, en el moment que sw[3] s'activi, ja podrem llegir el nombre.

A continuació disposem del codi en C del algorisme:

```
int Factorial(int n){
    int res;
    if ( n<2 ) res = 1;
    else res = n * Factorial(n-1);
    return res;
}
void main(){
    while ( sw[3] == 0 ) {}
    leds = Factorial( sw[2:0] );
}
```

FIGURA 216 – Exercici 6 enunciat. Codi en C del programa Factorial.

A partir d'aquí, fes els següents apartats:

a) Tradueix a llenguatge *CAL_assembler* el codi en C anterior. Pots fer ús de la crida de multiplicació implementada en l'exercici anterior.

Utilitza la implementació d'una pila per a no perdre la direcció de retorn en cada crida. La pila a de fer ús de les últimes posicions de la Memòria de Dades per a emmagatzemar els valors temporals i R14 és el punter al top.

b) Compila el teu programa i corregeix els errors fins que generi el fitxer binari **Program.bin.**

c) Crea un programa de test que simuli el comportament dels interruptor per a testejar el teu programa.

d) Simula el comportament del programa, i genera 3 cronogrames diferents:

- Un que mostri el valor dels dispositius d'entrada sortida i per tant que garanteixi el correcte comportament del factorial.
- Un que mostri el valor del registre R14 i les posicions de la Memòria de Dades sobre les que es mou la pila.
- Un que mostri el PC i el registres utilitzats per a fer la crida recursiva, per analitzar el seu funcionament.

SOLUCIÓ:

a) Una possible traducció pot ser:

```
etiq_main:
        LI R14,0
        LIH R14,1           //R14 = &topPila
        LI R10,1
        LIH R10,1           // R10 = &sw
        LI R0,8             // R0 = 0x0008
etiq_espera:
        LD R1,0(R10)        // R1 = sw
        AND R3,R1,R0
        BZ R3, etiq_espera

        LI R3,-1
        XOR R0,R0,R3        // R0 = 0xFFF7
        AND R1,R1,R0  //R1 = sw[2:0] Pas paràmetre

        LI R15,etiq_factorial
        LIH R15,etiq_factorial
        JR R15,R15,0

        ST -1(R10),R0       //leds = R0 = Retorn
etiq_fi:
        JMPI etiq_fi


etiq_multiplicacio:
        LI R0,0
etiq_while:
        BZ R2, etiq_fiwhile
        ADD R0,R0,R1
        ADDI R2,R2,-1
        JMPI etiq_while
etiq_fiwhile:
        JMP R15
```

```
etiq_factorial:
        // R0 = res
        ADDI R3,R1,-2
        BNEG R3,etiq_else

        //Crida recursiva
        ADDI R1,R1,-1       //Pas paràmetre
        ADDI R14,R14,-1     //Faig lloc en la pila
        ST 0(R14),R15       //Guardo direcció retorn

        LI R15,etiq_factorial
        LIH R15,etiq_factorial
        JR R15,R15,0
        //Valor retorn factorial en R0

        //crida multiplicació
        ADDI R1,R1,1        // Pas paràmetre n
        ADDI R2,R0,0        //Pas paràmetre fact

        LI R15,etiq_multiplicacio
        LIH R15,etiq_multiplicacio
        JR R15,R15,0
        //Resultat multiplicació queda en R0 = res

        LD R15,0(R14)       //Recupero direcció
retorn
        ADDI R14,R14,1      //Desfaig en la pila

        JMPI etiq_fiif
etiq_else:
        LI R0,1
etiq_fiif:
        JMP R15
```

FIGURA 217 – Exercici 6 solució a). Codi en CAL_assembler del programa Factorial.

b) Per aconseguir **program.bin** veure apartat: *Recursos per Exercicis – Exercici6 – b)*

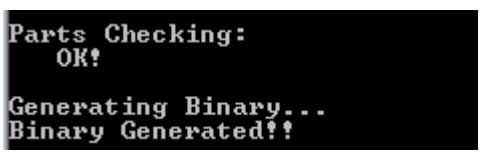


FIGURA 218 – Exercici 6 solució b). Compilació correcte del programa Factorial.

c)

```
module test_processador();
    reg clk,reset;
    reg [3:0] sw;
    wire [7:0] led;

    Processador Cal16 (clk,sw,reset,led);

    initial begin
        reset <= 1'b1;
        sw <= 4'h0;
        #3 reset <= 1'b0;
        #20 sw[2:0] <= 3'h4;
        #20 sw[3] <= 1'b1;
    end

    initial clk <= 1;
    always begin
        #1 clk <= ~clk;
    end
endmodule
```

FIGURA 219 – Exercici 6 solució c). Fitxer de test en Verilog per la simulació del programa Factorial.

d) A continuació trobem els diferents cronogrames segons el cas a analitzar.

- El següent cronograma mostra el valor dels dispositius d'entrada sortida i per tant que garanteix el correcte comportament del factorial:

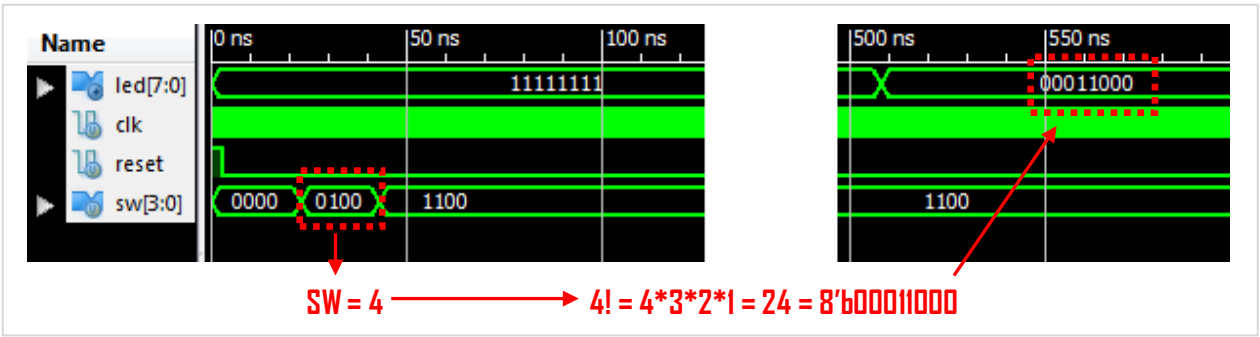


FIGURA 220 – Exercici 6 solució d). Cronograma que mostra el correcte funcionament del programa Factorial.

- El següent cronograma mostra el valor del registre R14 i les posicions de la Memòria de Dades sobre les que es mou la pila.

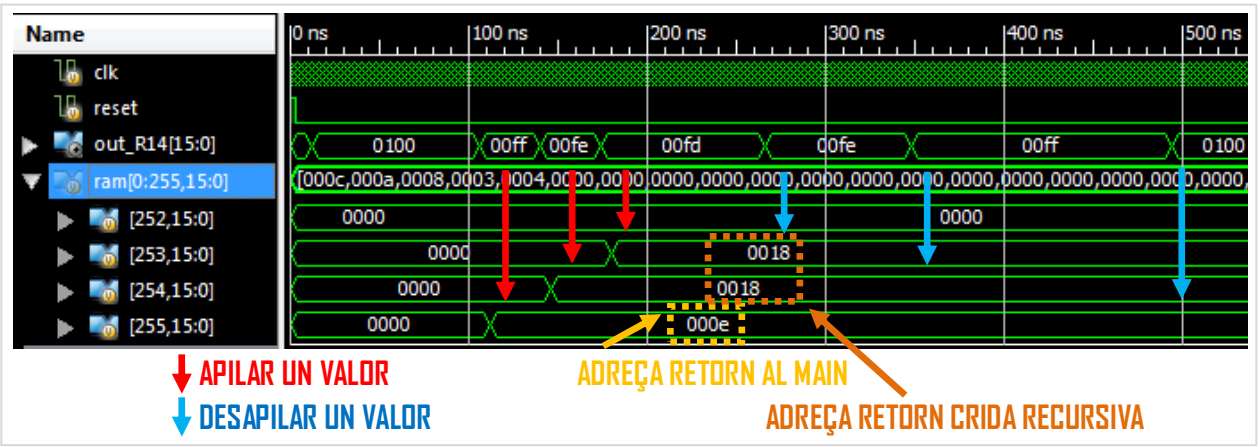


FIGURA 221 – Exercici 6 solució d). Cronograma que mostra l'evolució de la pila durant l'execució del programa Factorial.

- Un que mostri el PC i el registres utilitzats per a fer la crida recursiva, per analitzar el seu funcionament.

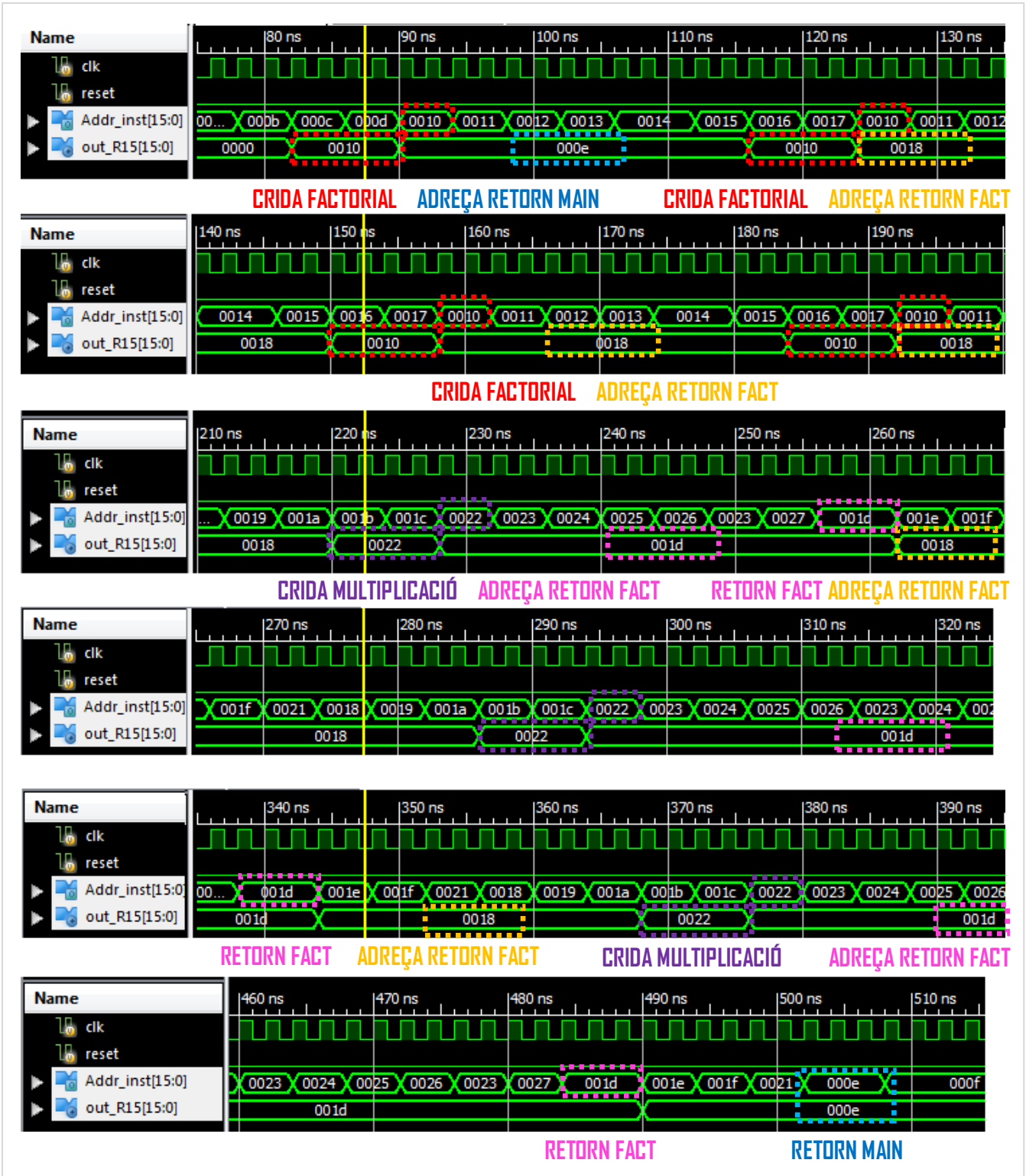


FIGURA 222 – Exercici 6 solució d). Cronograma que mostra l'evolució del PC durant l'execució del programa Factorial.

EXECUCIÓ DE PROGRAMES AMB FPGA

OBJECTIU

Aquest apartat té com a objectiu principal interactuar amb els dispositius d'entrada sortida de la PFGA a través dels programes en *CAL_assembler* i utilitzar recursos per a identificar errors d'execució. Per a demostrar el correcte funcionament de cada apartat s'adjunta un vídeo de l'execució dels programes.

EXERCICIS

EXERCICI 7: COMPTADOR LEDS

ENUNCIAT: Fes un programa que cada segon incrementi en 1 el valor emmagatzemat als leds. Inicialment el valor dels leds ha de ser 0, i l'efecte que es mostrarà amb l'execució serà d'un comptador de 0 a 255 en binari a través d'encendre i apagar els leds. El següent codi en C implementa el seu funcionament:

:

```
void espera1s(){
    for(i=0;i<(2^10);i++) {
        for(j=0;j<(2^11);j++){}
}

void main(){
    leds = 0;
    while (true){
        espera1s();
        leds++;
    }
}
```

FIGURA 223 – Exercici 7 enunciat. Codi en C del programa ComptadorLeds.

Més concretament:

a) Tradueix a llenguatge *CAL_assembler* el codi en C que implementa l'algorisme. Pots utilitzar la funció ja implementada en aquesta memòria de **espera1s().** Compila el codi i corregeix el errors fins que generi el fitxer binari **Program.bin.**

b) Modifica e fitxer **board.ucf** de configuració, per a permetre utilitzar el rellotge, els leds i el BTN_SOUTH de la FPGA. En cas de no tenir fitxer **board.ucf** veure apartat: *Recursos per Exercicis – Exercici7 – b)*

c) Fes els canvis necessaris també el teu processador per que utilitzi el rellotge generat per la FPGA, la sortida dels leds i el BTN_SOUTH com a senyal de reset.

d) Executa el teu programa amb la FPGA i comprova que els leds es comporten de la forma esperada.

SOLUCIÓ:

a) Una possible traducció pot ser:

```
etiq_main:
        LI R0,0
        LIH R0,1  //R10 = &leds
        LI R1,0
        ST 0(R0),R1         //leds = 0
etiq_infinit:
        LI R3,etiq_esperals
        LIH R3,etiq_esperals
        JR R15,R3,0         //espera 1 segon
        LD R1,0(R0)
        ADDI R1,R1,1
        ST 0(R0),R1
        JMPI etiq_infinit

etiq_esperals:
        LI R10,0  //R10=i=0
        LI R11,0
        LIH R11,-4 //R11=0xFC00=-1024
etiq_for1:
        ADD R9,R10,R11
        BZ R9, etiq_fifor1
        LI R12,0  //R12=j=0
        LI R13,0
        LIH R13,-8          //R13=0xF800=-2048
etiq_for2:
        ADD R9,R12,R13
        BZ R9, etiq_fifor2
        ADDI R12,R12,1
        JMPI etiq_for2
etiq_fifor2:
        ADDI R10,R10,1
        JMPI etiq_for1
etiq_fifor1:
        JMP R15
```

FIGURA 224 – Exercici 7 solució a). Codi en CAL_assembler del programa ComptadorLeds.

*Per aconseguir **program.bin** veure apartat: *Recursos per Exercicis – Exercici7 – a)*

b) Descomentar les següents línees del fitxer **board.ucf**. En cas de no tenir fitxer **board.ucf** veure apartat: *Recursos per Exercicis – Exercici7 – b)*

```
# ==== Pushbuttons (BTN) ====
#NET "BTN_EAST"  LOC = "H13" | IOSTANDARD = LVTTL  | PULLDOWN ;
#NET "BTN_NORTH" LOC = "V4"  | IOSTANDARD = LVTTL  | PULLDOWN ;
NET "BTN_SOUTH" LOC = "K17" | IOSTANDARD = LVTTL  | PULLDOWN ;
#NET "BTN_WEST"  LOC = "D18" | IOSTANDARD = LVTTL  | PULLDOWN ;


# ==== Clock inputs (CLK) ====
NET "clk" LOC = "C9" | IOSTANDARD = LVTTL;
#      Define clock period for 50 MHz oscillator (40%/60% duty-cycle)
#NET "clk" PERIOD = 20.0ns HIGH 40%;
NET "clk" PERIOD = 30.0ns HIGH 50%;
#NET "CLK_AUX"   LOC = "B8"  | IOSTANDARD = LVCMOS33 ;
#NET "CLK_SMA"   LOC = "A10" | IOSTANDARD = LVCMOS33 ;
```

FIGURA 225 – Exercici 7 solució b). Fitxer board.ucf pel programa ComptadorLeds.

* Continua a la següent pàgina

```
# ==== Discrete LEDs (LED) ====
# These are shared connections with the FX2 connector
NET "LED<0>"   LOC = "F12" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<1>"   LOC = "E12" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<2>"   LOC = "E11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<3>"   LOC = "F11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<4>"   LOC = "C11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<5>"   LOC = "D11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<6>"   LOC = "E9"  | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<7>"   LOC = "F9"  | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
```

FIGURA 225 – Exercici 7 solució b). Fitxer board.ucf pel programa ComptadorLeds.

c)

```
module Processador(input clk, input [3:0] sw, input BTN_SOUTH, output [7:0] led);
    wire [15:0] Instruccio,Addr_inst,Nova_inst;
    wire [15:0] DataInMem,DataOutMem,DataOutMem0,DataOutMem1,DataOutMem2,AddrMem;
    wire [7:0] s_leds;
    wire [3:0] s_sw;
    wire wrMemData,fetch,mem;
    wire reset;

    UnitatProcesCentral CPU (Instruccio,DataOutMem,clk,reset,DataInMem,AddrMem,Addr_inst,wrMemData,mem);

    assign reset = BTN_SOUTH;
    assign led = s_leds;
    /************************************** ESPAI DE MEMÒRIA **************************************/
        /************************************** INSTRUCCIONS ***************************************/
        Rom MemInst (clk,Addr_inst[11:0],Instruccio);
        /************************************** FI INSTRUCCIONS ************************************/

        /************************************** DADES **********************************************/
        Ram MemDades (clk,AddrMem[7:0],DataInMem,DataOutMem0,wrMemData,mem & !AddrMem[8]);
        /************************************** FI DADES *******************************************/

        /************************************** E/S ************************************************/
        RegLeds ES_leds (clk,DataInMem,DataOutMem1,s_leds,wrMemData,(AddrMem[8:0] == 16'h0100) & mem);
        RegSwitches ES_switches (clk,sw,DataOutMem2);
        /************************************** FI E/S *********************************************/
    mux4_1_16bit MUX_PROC (16'h0000,DataOutMem2,DataOutMem1,DataOutMem0,
                           {AddrMem[8:0] == 16'h0101,AddrMem[8:0] == 16'h0100},DataOutMem);
    /************************************** FI ESPAI DE MEMÒRIA ************************************/
endmodule
```

FIGURA 226 – Exercici 7 solució c). Assignació de dispositius a l'espai d'entrada sortida del processador pel programa ComptadorLeds.

d) Per a comprovar el correcte funcionament del programa veure apartat: *Recursos per Exercicis – Exercici7 – d)*

EXERCICI 8: MOSTRA VALORS D'UN VECTOR

ENUNCIAT: Fes un programa que donat un vector de 16 posicions emmagatzemat a la Memòria de Dades a partir de la direcció 0x001, mostri a través dels leds el valor de la posició codificada en binari pels interruptors.

Concretament a cada canvi del nombre codificat pels interruptors, mostrar el valor que conté la posició del vector codificada pels interruptors. Per exemple si els interruptors codifiquen el 4, els leds hauran de prendre el valor de vec[4]. Per detectar els canvis dels interruptors es disposa d'una variable ant, amb direcció 0 a la Memòria de Dades, on guardar l'últim valor llegit dels interruptors.

El nombre màxim a codificar a través dels interruptors és 15, així que no t'has de preocupar per que l'índex del vector no es pot sortir de rang.

Recorda que entre canvi i canvi dels interruptors és important que esperis un temps per permetre que les senyals estiguin estables. Pots utilitzar els programes que hagis programat en exercicis anteriors en cas que els necessitis. El següent codi en C implementa el seu funcionament:

```
MemData[0x000] ant = 0;
MemData[0x001] vec[16] = {3,9,1,38,4,
        4,25,10,11,8,7,21,12,2,55,5};

void main(){
    while(true){
        ant = sw;
        leds = vec[ant];
        while (sw == ant) espera1s();
    }
}
void espera1s(){
    for(i=0;i<(2^10);i++) {
        for(j=0;j<(2^11);j++){}
}
```

FIGURA 227 – Exercici 8 enunciat. Codi en C del programa Valors d'un vector.

Més concretament:

a) Tradueix a llenguatge *CAL_assembler* el codi en C que implementa l'algorisme. Pots utilitzar la funció ja implementada en aquesta memòria de **espera1s().** Compila el codi i corregeix el errors fins que generi el fitxer binari **Program.bin.**

b) Modifica e fitxer **board.ucf** de configuració, per a permetre utilitzar el rellotge, els leds, els interruptors i el BTN_SOUTH de la FPGA. En cas de no tenir fitxer **board.ucf** veure apartat: *Recursos per Exercicis – Exercici8 – b)*

c) Fes els canvis necessaris també el teu processador per que utilitzi el rellotge generat per la FPGA, l'entrada dels interruptors, la sortida dels leds i el BTN_SOUTH com a senyal de reset.

d) Executa el teu programa amb la FPGA i comprova que els leds es comporten de la forma esperada.

SOLUCIÓ:

a) Una possible traducció pot ser:

```
etiq_main:
        LI R8,-1        //R8=-1
        LI R0,0         //R0=&ant
        LI R1,0
        LIH R1,1        //R1=&leds
        LI R2,1         //R2=&vec
        LD R3,1(R1)     //R3=sw
etiq_inf:
        ST 0(R0),R3     //ant=sw
        ADD R4,R2,R3    //R4=&vec[ant]
        LD R4,0(R4)     //R4=vec[ant]
        ST 0(R1),R4     //leds=vec[ant]
        LD R4,0(R0)     //R4=ant
        XOR R4,R4,R8
        ADDI R4,R4,1    //R4=-ant
etiq_enquesta:
        LI R15,etiq_esperals
        LIH R15,etiq_esperals
        JR R15,R15,0    //espera 1 segon
        LD R3,1(R1)     //R3=sw
        ADD R5,R3,R4
        BZ R5, etiq_enquesta
        JMPI etiq_inf
etiq_esperals:
        LI R10,0  //R10=i=0
        LI R11,0
        LIH R11,-4 //R11=0xFC00=-1024
etiq_for1:
        ADD R9,R10,R11
        BZ R9, etiq_fifor1
        LI R12,0  //R12=j=0
        LI R13,0
        LIH R13,-8          //R13=0xF800=-2048
etiq_for2:
        ADD R9,R12,R13
        BZ R9, etiq_fifor2
        ADDI R12,R12,1
        JMPI etiq_for2
etiq_fifor2:
        ADDI R10,R10,1
        JMPI etiq_for1
etiq_fifor1:
        JMP R15
```

FIGURA 228 – Exercici 8 solució a). Codi en CAL_assembler del programa Valors d'un vector.

*Per aconseguir **program.bin** veure apartat: *Recursos per Exercicis – Exercici8 – a)*

b) Descomentar les següents línees del fitxer **board.ucf**. En cas de no tenir fitxer **board.ucf** veure apartat: *Recursos per Exercicis – Exercici8 – b)*

```
# ==== Pushbuttons (BTN) ====
#NET "BTN_EAST"  LOC = "H13" | IOSTANDARD = LVTTL  | PULLDOWN ;
#NET "BTN_NORTH" LOC = "V4"  | IOSTANDARD = LVTTL  | PULLDOWN ;
NET "BTN_SOUTH" LOC = "K17" | IOSTANDARD = LVTTL  | PULLDOWN ;
#NET "BTN_WEST"  LOC = "D18" | IOSTANDARD = LVTTL  | PULLDOWN ;


# ==== Clock inputs (CLK) ====
NET "clk" LOC = "C9" | IOSTANDARD = LVTTL;
#      Define clock period for 50 MHz oscillator (40%/60% duty-cycle)
#NET "clk" PERIOD = 20.0ns HIGH 40%;
NET "clk" PERIOD = 30.0ns HIGH 50%;
#NET "CLK_AUX"   LOC = "B8"  | IOSTANDARD = LVCMOS33 ;
#NET "CLK_SMA"   LOC = "A10" | IOSTANDARD = LVCMOS33 ;
```

FIGURA 229 – Exercici 8 solució b). Fitxer board.ucf del programa Valors d'un vector.

* Continua a la següent pàgina

```
# ==== Discrete LEDs (LED) ====
# These are shared connections with the FX2 connector
NET "LED<0>"  LOC = "F12" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<1>"  LOC = "E12" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<2>"  LOC = "E11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<3>"  LOC = "F11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<4>"  LOC = "C11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<5>"  LOC = "D11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<6>"  LOC = "E9"  | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<7>"  LOC = "F9"  | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;

# ==== Slide Switches (SW) ====
NET "SW<0>"  LOC = "L13" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<1>"  LOC = "L14" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<2>"  LOC = "H18" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<3>"  LOC = "N17" | IOSTANDARD = LVTTL | PULLUP ;
```

FIGURA 229 – Exercici 8 solució b). Fitxer board.ucf del programa Valors d'un vector.

c)

```
module Processador(input clk, input [3:0] sw, input BTN_SOUTH, output [7:0] led);
    wire [15:0] Instruccio,Addr_inst,Nova_inst;
    wire [15:0] DataInMem,DataOutMem,DataOutMem0,DataOutMem1,DataOutMem2,AddrMem;
    wire wrMemData,fetch,mem;
    wire reset;

    UnitatProcesCentral CPU (Instruccio,DataOutMem,clk,reset,DataInMem,AddrMem,Addr_inst,wrMemData,mem);

    assign reset = BTN_SOUTH;
    /************************************ ESPAI DE MEMÒRIA ************************************/
        /************************************ INSTRUCCIONS ************************************/
        Rom MemInst (clk,Addr_inst[11:0],Instruccio);
        /************************************ FI INSTRUCCIONS ************************************/

        /************************************ DADES ************************************/
        Ram MemDades (clk,AddrMem[7:0],DataInMem,DataOutMem0,wrMemData,mem & !AddrMem[8]);
        /************************************ FI DADES ************************************/

        /************************************ E/S ************************************/
        RegLeds ES_leds (clk,DataInMem,DataOutMem1,led,wrMemData,(AddrMem[8:0] == 16'h0100) & mem);
        RegSwitches ES_switches (clk,sw,DataOutMem2);
        /************************************ FI E/S ************************************/
    mux4_1_16bit MUX_PROC (16'h0000,DataOutMem2,DataOutMem1,DataOutMem0,
                           {AddrMem[8:0] == 16'h0101,AddrMem[8:0] == 16'h0100},DataOutMem);
    /************************************ FI ESPAI DE MEMÒRIA ************************************/
endmodule
```

FIGURA 230 – Exercici 8 solució c). Assignació de dispositius a l'espai d'entrada sortida del processador pel programa Valors d'un vector.

d) Per a comprovar el correcte funcionament del programa veure apartat: *Recursos per Exercicis – Exercici8 – d)*

EXERCICI 9: CONÈIXER L'ESTAT D'UN PROGRAMA

ENUNCIAT: Programa la FPGA amb l'exercici 6, Factorial d'un nombre, el qual obtenia el nombre codificat pels interruptors de menys pes i mostrava pels leds el valor del seu factorial.

Més que voler veure el seu correcte funcionament, volem poder consultar l'estat en el que es troba a mitja execució, per això modificarem la programació del processador, per que en els leds es mostri en funció dels interruptors el següent:

Si **sw[3] == 1** llavors **leds = factorial (sw[2:0])**
Si **sw[3] == 0** llavors
    Si **sw[2:0]==3'b000** llavors **leds = Instrucció[7:0]**
    Si **sw[2:0]==3'b001** llavors **leds = Instrucció[15:8]**
    Si **sw[2:0]==3'b010** llavors **leds = Addr_inst[7:0]**
    Si **sw[2:0]==3'b011** llavors **leds = Addr_inst [15:8]**
    Si **sw[2:0]==3'b100** llavors **leds = {1'b1, 2'b00,fetch,1'b0,exec, 1'b0,mem}**
    Si **sw[2:0]==3'b101** llavors **leds = 8'b11111111**
    Si **sw[2:0]==3'b110** llavors **leds = 8'b10101010**
    Si **sw[2:0]==3'b111** llavors **leds = 8'b10101010**

FIGURA 231 – Exercici 9 enunciat. Pseudocodi que especifica el comportament del programa Conèixer l'estat d'un programa.

Primer comprovarem el l'exercici Factorial d'un nombre funciona correctament a la FPGA:

a) Compila el codi del exercici 6, Factorial d'un nombre i corregeix el errors fins que generi el fitxer binari **Program.bin.**

b) Modifica e fitxer **board.ucf** de configuració, per a permetre utilitzar el rellotge, els leds, els interruptors i el BTN_SOUTH de la FPGA. En cas de no tenir fitxer **board.ucf** veure apartat: *Recursos per Exercicis – Exercici9 – b)*

c) Fes els canvis necessaris també el teu processador per que utilitzi el rellotge generat per la FPGA, la sortida dels leds, l'entrada dels interruptors i el BTN_SOUTH com a senyal de reset.

d) Executa el teu programa amb la FPGA i comprova que els leds es comporten de la forma esperada.

Ara generarem el codi de consulta de l'estat del processador:

e) Modifica el processador per que les senyals de exec i fetch arribin al fitxer arrel de **processador.v.** Cal que les afegeixis de sortida del bloc UC i les propaguis a través del bloc CPU fins que t'arribin al nivell que vols.

f) Afegeix un nou bus per recollir el valor del registre del leds del processador i una nova assignació a la sortida de leds del fitxer processador.v, que pregui valor en funció del sw tal i com s'ha determinar en l'especificació anterior.

g) Comprova el correcte funcionament programant a la FPGA amb el processador modificat.

SOLUCIÓ:

a) Per veure la traducció del codi veure apartat: *Simulació de Programes en CAL_assembler – Exercicis – Exercici6 – a*
Per aconseguir **program.bin** veure apartat: *Recursos per Exercicis – Exercici6 – b)*

b) Descomentar les següents línees del fitxer **board.ucf**. En cas de no tenir fitxer **board.ucf** veure apartat: *Recursos per Exercicis – Exercici9 – b)*

```
# ==== Pushbuttons (BTN) ====
#NET "BTN_EAST"  LOC = "H13" | IOSTANDARD = LVTTL  | PULLDOWN ;
#NET "BTN_NORTH" LOC = "V4"  | IOSTANDARD = LVTTL  | PULLDOWN ;
NET "BTN_SOUTH" LOC = "K17" | IOSTANDARD = LVTTL  | PULLDOWN ;
#NET "BTN_WEST"  LOC = "D18" | IOSTANDARD = LVTTL  | PULLDOWN ;


# ==== Clock inputs (CLK) ====
NET "clk" LOC = "C9" | IOSTANDARD = LVTTL;
#      Define clock period for 50 MHz oscillator (40%/60% duty-cycle)
#NET "clk" PERIOD = 20.0ns HIGH 40%;
NET "clk" PERIOD = 30.0ns HIGH 50%;
#NET "CLK_AUX"   LOC = "B8"  | IOSTANDARD = LVCMOS33 ;
#NET "CLK_SMA"   LOC = "A10" | IOSTANDARD = LVCMOS33 ;

# ==== Discrete LEDs (LED) ====
# These are shared connections with the FX2 connector
NET "LED<0>"  LOC = "F12" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<1>"  LOC = "E12" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<2>"  LOC = "E11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<3>"  LOC = "F11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<4>"  LOC = "C11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<5>"  LOC = "D11" | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<6>"  LOC = "E9"  | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;
NET "LED<7>"  LOC = "F9"  | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 8 ;

# ==== Slide Switches (SW) ====
NET "SW<0>"  LOC = "L13" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<1>"  LOC = "L14" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<2>"  LOC = "H18" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<3>"  LOC = "N17" | IOSTANDARD = LVTTL | PULLUP ;
```

FIGURA 232 – Exercici 9 solució b). Fitxer board.ucf del programa Conèixer l'estat d'un programa.

c)

```
module Processador(input clk, input [3:0] sw, input BTN_SOUTH, output [7:0] led);
    wire [15:0] Instruccio,Addr_inst,Nova_inst;
    wire [15:0] DataInMem,DataOutMem,DataOutMem0,DataOutMem1,DataOutMem2,AddrMem;
    wire wrMemData,fetch,mem;
    wire reset;

    UnitatProcesCentral CPU (Instruccio,DataOutMem,clk,reset,DataInMem,AddrMem,Addr_inst,wrMemData,mem);

    assign reset = BTN_SOUTH;
    /************************************** ESPAI DE MEMÒRIA **************************************/
        /************************************** INSTRUCCIONS ***************************************/
        Rom MemInst (clk,Addr_inst[11:0],Instruccio);
        /************************************** FI INSTRUCCIONS ************************************/

        /************************************** DADES **********************************************/
        Ram MemDades (clk,AddrMem[7:0],DataInMem,DataOutMem0,wrMemData,mem & !AddrMem[8]);
        /************************************** FI DADES *******************************************/

        /************************************** E/S ************************************************/
        RegLeds ES_leds (clk,DataInMem,DataOutMem1,led,wrMemData,(AddrMem[8:0] == 16'h0100) & mem);
        RegSwitches ES_switches (clk,sw,DataOutMem2);
        /************************************** FI E/S *********************************************/
    mux4_1_16bit MUX_PROC (16'h0000,DataOutMem2,DataOutMem1,DataOutMem0,
                           {AddrMem[8:0] == 16'h0101,AddrMem[8:0] == 16'h0100},DataOutMem);
    /************************************** FI ESPAI DE MEMÒRIA ************************************/
endmodule
```

FIGURA 233 – Exercici 9 solució c). Assignació de dispositius a l'espai d'entrada sortida del processador pel programa Conèixer l'estat d'un programa.

d) Per a comprovar el correcte funcionament del programa veure apartat: *Recursos per Exercicis – Exercici9 – d)*

e) Cal modificar els següents fitxers:

```
module UnitatControl(Instruccio,clk,reset,
                     OutOffset,Asel,Bsel,Dsel,op,alu_rotr,b_off,up_mem,jmp,bz,bneg,
                        w_Rd,nova_inst,mem,wrMemData,fetch,exec);

   input [15:0] Instruccio;
   input clk,reset;

   output [15:0] OutOffset;
   output [3:0] Asel,Bsel,Dsel;
   output [2:0] op;
   output alu_rotr,b_off,up_mem,jmp,bz,bneg,w_Rd,nova_inst;
   output mem,wrMemData;
   output fetch,exec;

   wire e0,e_mes0,e1,e_mes1;
   wire RDenRB,RAenRB,RAenRD,ld_st;
   wire [3:0] RA,RB,RD;
   wire [16:0] paraulaControl;
```

FIGURA 234 – Exercici 9 solució e). Propagació senyals fetch i exec de sde UC pel programa Conèixer l'estat d'un programa.

```
module UnitatProcesCentral(Instruccio,DataOutMem,clk,reset,
                           DataInMem,AddrMem,Addr_inst,wrMemData,mem,fetch,exec);

   input clk, reset;
   input [15:0] Instruccio,DataOutMem;

   output wrMemData,mem;
   output fetch,exec;
   output [15:0] DataInMem,AddrMem,Addr_inst;

   wire [15:0] Offset;
   wire [3:0] RA,RB,RD;
   wire [2:0] op;
   wire alu_rotr,b_off,up_mem,jmp,jr,bz,bneg,w_Rd,nova_inst;
   wire Cout;

   UnitatControl UC (Instruccio,clk,reset,Offset,RA,RB,RD,op,alu_rotr,b_off,
                     up_mem,jmp,bz,bneg,w_Rd,nova_inst,mem,wrMemData,fetch,exec);
   UnitatProces UP (Offset,RA,RB,RD,op,clk,reset,alu_rotr,b_off,up_mem,jmp,bz,
                    bneg,w_Rd,nova_inst,DataOutMem,DataInMem,AddrMem,Addr_inst);

endmodule
```

FIGURA 235 – Exercici 9 solució e). Propagació senyals fetch i exec des de CPU pel programa Conèixer l'estat d'un programa.

```
module Processador(input clk, input [3:0] sw, input BTN_SOUTH, output [7:0] led);

   wire [15:0] Instruccio,Addr_inst,Nova_inst;
   wire [15:0] DataInMem,DataOutMem,DataOutMem0,DataOutMem1,DataOutMem2,AddrMem;
   wire [7:0] s_leds;
   wire [3:0] s_sw;
   wire wrMemData,fetch,exec,mem;
   wire reset;

   UnitatProcesCentral CPU (Instruccio,DataOutMem,clk,reset,DataInMem,
                            AddrMem,Addr_inst,wrMemData,mem,fetch,exec);
```

FIGURA 236 – Exercici 9 solució e). Propagació senyals fetch i exec fina al Processador pel programa Conèixer l'estat d'un programa.

f) En el fitxer **processador.v** afegir:

```
module Processador(input clk, input [3:0] sw, input BTN_SOUTH, output [7:0] led);

   wire [15:0] Instruccio,Addr_inst,Nova_inst;
   wire [15:0] DataInMem,DataOutMem,DataOutMem0,DataOutMem1,DataOutMem2,AddrMem;
   wire wrMemData,fetch,exec,mem;
   wire reset;
   wire [7:0] s_leds;

   UnitatProcesCentral CPU (Instruccio,DataOutMem,clk,reset,DataInMem,
                            AddrMem,Addr_inst,wrMemData,mem,fetch,exec);

   assign reset = BTN_SOUTH;
   assign led = sw[3] ? s_leds :
      (sw[2] ?
         (sw[1] ? 8'b10101010
         : (sw[0] ? 8'b11111111 : {1'b1, 2'b00,fetch,1'b0,exec, 1'b0,mem} )
         )
      :
         (sw[1] ?
            (sw[0] ? Addr_inst[15:8] : Addr_inst[7:0] )
         :
            (sw[0] ? Instruccio[15:8] : Instruccio[7:0] )
         )
      );
   /************************************** ESPAI DE MEMÒRIA ***************************************/

   /************************************** INSTRUCCIONS ***************************************/
   Rom MemInst (clk,Addr_inst[11:0],Instruccio);
   /************************************** FI INSTRUCCIONS ***************************************/

   /************************************** DADES ***************************************/
   Ram MemDades (clk,AddrMem[7:0],DataInMem,DataOutMem0,wrMemData,mem & !AddrMem[8]);
   /************************************** FI DADES ***************************************/

   /************************************** E/S ***************************************/
   RegLeds ES_leds (clk,DataInMem,DataOutMem1,s_leds,wrMemData, (AddrMem[8:0] == 16'h0100) & mem);

   RegSwitches ES_switches (clk,sw,DataOutMem2);
   /************************************** FI E/S ***************************************/

   mux4_1_16bit MUX_PROC (16'h0000,DataOutMem2,DataOutMem1,DataOutMem0,
                          {AddrMem[8:0] == 16'h0101,AddrMem[8:0] == 16'h0100},DataOutMem);
   /************************************** FI ESPAI DE MEMÒRIA ***************************************/
endmodule
```

FIGURA 237 – Exercici 9 solució f). Assignar valor dels leds pel programa Conèixer l'estat d'un programa.

g) Per a comprovar el correcte funcionament del programa veure apartat: *Recursos per Exercicis – Exercici9 – g)*

AMPLIACIÓ DEL PROCESSADOR

OBJECTIU

Aquest apartat té com a objectiu principal implementar nous blocs per afegir funcionalitats al processador. Per aconseguir-ho, cal tenir una bon coneixement de com programar en Verilog, de tots els busos, blocs i senyals que componen el *CAL16*, ja que al realitzar les modificacions no variï el comportament actual de les instruccions de les que ja disposa el processador. Per cada modificació, els canvis en el compilador ja es donen fets.

EXERCICIS

EXERCICI 10: INSTRUCCIÓ SUB

ENUNCIAT: Amplia el processador *CAL16* de manera que incorporis la instrucció SUB Rd,Ra,Rb. Aquesta realitza la resta entre Ra menys Rb deixant el resultat en Rd, sent tots nombres de 16 bits. La instrucció SUB tindrà el codi d'operació 15.

SUB Rd,Ra,Rb	**Rd = Ra – Rb**	**Codi Operació = 4'b1111**
	Codificació:	**1111 aaaa dddd bbbb**

FIGURA 238 – Exercici 10 enunciat. Especificació instrucció SUB.

Segueix els següents passos:

a) Crea un nou fitxer verilog en el directori de Blocs Combinacionals del processador i implementa el bloc **SUB_16b,** si vols pots basar-te en el bloc **SUB_4b** de l'exercici 2.

b) Modifica el bloc ALU per que pugui realitzar la operació de resta. Crear un fitxer de test pel bloc ALU i comprova el seu correcte funcionament.

c) Assigna la paraula de control corresponent a la operació SUB, al bloc UC. Recorda que el seu codi d'operació és el 15.

d) Modifica els fitxers del compilador amb els que se't donen a l'apartat: *Recursos per Exercicis – Exercici10 – d).* Compila el compilador amb la comanada make.

e) Implementa un programa molt simple que realitzi una resta amb la instrucció SUB i simula el seu comportament per confirmar el correcte funcionament.

SOLUCIÓ:

a)

```
module SUB_16b(A,B,W,b,ovf);
    input [15:0] A,B;
    output [15:0] W;
    output b,ovf;

    wire b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14;
    wire [15:0] A_n;

    not_1bit not0(A[0],A_n[0]);
    not_1bit not1(A[1],A_n[1]);
    not_1bit not2(A[2],A_n[2]);
    not_1bit not3(A[3],A_n[3]);
    not_1bit not4(A[4],A_n[4]);
    not_1bit not5(A[5],A_n[5]);
    not_1bit not6(A[6],A_n[6]);
    not_1bit not7(A[7],A_n[7]);
    not_1bit not8(A[8],A_n[8]);
    not_1bit not9(A[9],A_n[9]);
    not_1bit not10(A[10],A_n[10]);
    not_1bit not11(A[11],A_n[11]);
    not_1bit not12(A[12],A_n[12]);
    not_1bit not13(A[13],A_n[13]);
    not_1bit not14(A[14],A_n[14]);
    not_1bit not15(A[15],A_n[15]);

    fullAdder bit0 (A_n[0],B[0],1'b1,W[0],b0);
    fullAdder bit1 (A_n[1],B[1],b0,W[1],b1);
    fullAdder bit2 (A_n[2],B[2],b1,W[2],b2);
    fullAdder bit3 (A_n[3],B[3],b2,W[3],b3);
    fullAdder bit4 (A_n[4],B[4],b3,W[4],b4);
    fullAdder bit5 (A_n[5],B[5],b4,W[5],b5);
    fullAdder bit6 (A_n[6],B[6],b5,W[6],b6);
    fullAdder bit7 (A_n[7],B[7],b6,W[7],b7);
    fullAdder bit8 (A_n[8],B[8],b7,W[8],b8);
    fullAdder bit9 (A_n[9],B[9],b8,W[9],b9);
    fullAdder bit10 (A_n[10],B[10],b9,W[10],b10);
    fullAdder bit11 (A_n[11],B[11],b10,W[11],b11);
    fullAdder bit12 (A_n[12],B[12],b11,W[12],b12);
    fullAdder bit13 (A_n[13],B[13],b12,W[13],b13);
    fullAdder bit14 (A_n[14],B[14],b13,W[14],b14);
    fullAdder bit15 (A_n[15],B[15],b14,W[15],b15);

    not_1bit not16 (b15,b);
    xor_1b xor_ovf (b15,b14,ovf);
endmodule
```

FIGURA 239 – Exercici 10 solució a). Implementació en Verilog del bloc SUB_16b.

b)

```
module ALU(A,B,op,Cin,W,z,neg);
    input [15:0] A,B;
    input[2:0] op;
    input Cin;
    output [15:0] W;
    output z,neg;

    wire [15:0] W_and, W_or, W_xor, W_add, W_muxComb, W_sub;
    wire Cout,b,ovf;

    AND_16b AND (A,B,W_and);
    OR_16b OR (A,B,W_or);
    XOR_16b XOR (A,B,W_xor);
    ADD_16b ADD (A,B,Cin,W_add,Cout);
    SUB_16b SUB (A,B,W_sub,b,ovf);

    mux8_1_16bit MUX_BLOCS_COMBINACIONALS (B, W_sub, {B[7:0],A[7:0]},
                                B, W_add, W_xor, W_or, W_and, op, W);

    assign neg = W[15];
    DetectorZero_16bit DET_Z (W,z);
endmodule
```

FIGURA 240 – Exercici 10 solució b). Implementació en Verilog de l'incorporació del bloc SUB_16b al bloc ALU del processador.

```
module testALU;
    reg [15:0] A,B;
    reg [2:0] op;
    reg Cin;

    wire [15:0] W;
    wire z,neg;

    ALU uut (.A(A),.B(B),.op(op),.Cin(Cin),.W(W),.z(z),.neg(neg));

    initial begin
        A = 0;
        B = 0;
        op = 3'b110;
        Cin = 0;
        #10 A = 16'h0009;
        B = 16'hffff;
        #10 A = 16'h1234;
        B = 16'h4321;
        #10 A = 16'hA549;
        B = 16'hC009;
        #10 A = 16'h8888;
        B = 16'h1111;
    end
endmodule
```

FIGURA 241 – Exercici 10 solució b). Implementació en Verilog del fitxer de test pel bloc ALU amb el nou bloc SUB incorporat.

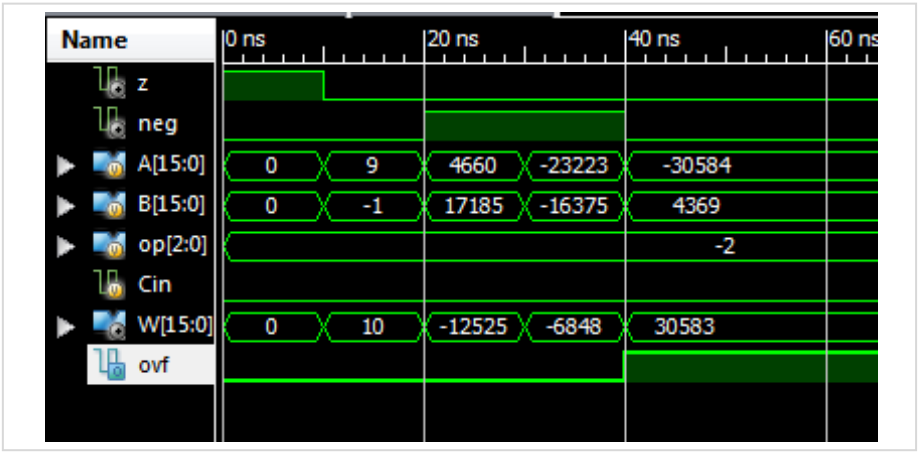


FIGURA 241 – Exercici 10 solució b). Cronograma que mostra el funcionament del fitxer de test pel bloc ALU amb el nou bloc SUB incorporat.

c)

```
reg [15:0] RomParaulaControl [15:0];
initial begin
   //op,op,op,alu_rotr,b_off,off4_off8,off12,jmp,bz,bneg,
   //wrMem,up_mem,RAenRD,RdenRB,RAenRB,wRd
   RomParaulaControl[0] = {18'b0000000000000001};  //AND
   RomParaulaControl[1] = {18'b0010000000000001};  //OR
   RomParaulaControl[2] = {18'b0100000000000001};  //XOR
   RomParaulaControl[3] = {18'b0110000000000001};  //ADD
   RomParaulaControl[4] = {18'b0110100000000001};  //ADDI
   RomParaulaControl[5] = {18'b0001000000000001};  //ROTR
   RomParaulaControl[6] = {18'b0110100000010001};  //LD
   RomParaulaControl[7] = {18'b0110100000100100};  //ST
   RomParaulaControl[8] = {18'b1000110000001001};  //LI
   RomParaulaControl[9] = {18'b1010110000001001};  // LIH
   RomParaulaControl[10] = {18'b1000010001000010}; //BNEG
   RomParaulaControl[11] = {18'b1000010010000010}; //BZ
   RomParaulaControl[12] = {18'b1000000100000011}; //JR
   RomParaulaControl[13] = {18'b1000000100000010}; //JMP
   RomParaulaControl[14] = {18'b1000101100000000}; //JMPI
   RomParaulaControl[15] = {18'b1100000000000001}; //SUB
end
```

FIGURA 242 – Exercici 10 solució c). Implementació en Verilog de la rom paraula de control amb la nova instrucció SUB incorporada.

d) Veure apartat: *Recursos per Exercicis – Exercici10 – d).*

e)

LI R1,9
LI R2,5
SUB R3,R1,R2

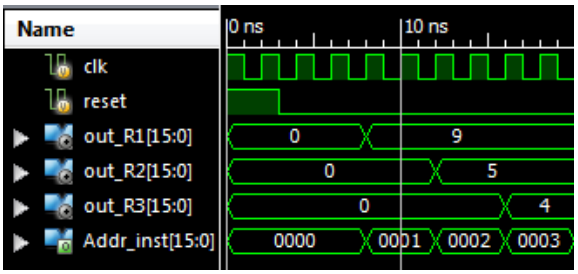


FIGURA 243 – Exercici 10 solució e). Cronograma que mostra el funcionament d'un programa en CAL_assembler que utilitza la instrucció SUB.

EXERCICI 11: FUSIÓ BNEG & BZ

ENUNCIAT: Modifica el processador *CAL16* per a que disposi de les dues instruccions de salt condicional i relatiu al PC, fent ús d'un codi d'operació per a les dues, amb codi 10, deixant el codi 11 lliure per a noves instruccions. Per aconseguir-ho sacrificarem un bit de l'immediat que forma l'operació, el de més pes, el qual s'utilitzarà com a identificador de la condició:

BNEG: 1010 aaaa 0 ooooooo

BZ: 1010 aaaa 1 ooooooo

FIGURA 244 – Exercici 11 enunciat. Especificació de la nova codificació per les instruccions BNEG i BZ.

Segueix els següents passos:

a) Modifica el bloc UC per que les senyals bneg i bz ja no es generin a través de la rom de paraula de control, sinó a partir dels bits que codifiquem la instrucció. Pel que fa al immediat el tractarem com un immediat de 8 bits, l'únic que si es tracta d'una instrucció d'aquest tipus haurem d'assegurar-nos d'estendre-li el signe i no agafar el nou bit que hem definit per a diferenciar els salts.

b) Si et cal, fes les modificacions que creguis oportunes a la UP.

c) Modifica els fitxers del compilador amb els que se't donen a l'apartat: *Recursos per Exercicis – Exercici11 – c)*. Compila el compilador amb la comanada make.

d) Prova d'executar el programa factorial, exercici 6 de la col·lecció, per a comprovar el correcte funcionament de les dues instruccions.

SOLUCIÓ:

a) A continuació es mostra una possible modificació del fitxer UC.v. Primer cal modificar la rom paraula de control de la següent manera:

```
reg [15:0] RomParaulaControl [15:0];
initial begin
   //op,op,op,alu_rotr,b_off,off4_off8,off12,jmp,b_cond,
   //wrMem,up_mem,RAenRD,RdenRB,RAenRB,wRd
   RomParaulaControl[0] =  {18'b000000000000001};  //AND
   RomParaulaControl[1] =  {18'b001000000000001};  //OR
   RomParaulaControl[2] =  {18'b010000000000001};  //XOR
   RomParaulaControl[3] =  {18'b011000000000001};  //ADD
   RomParaulaControl[4] =  {18'b011010000000001};  //ADDI
   RomParaulaControl[5] =  {18'b000100000000001};  //ROTR
   RomParaulaControl[6] =  {18'b011010000010001};  //LD
   RomParaulaControl[7] =  {18'b011010000100100};  //ST
   RomParaulaControl[8] =  {18'b100011000001001};  //LI
   RomParaulaControl[9] =  {18'b101011000001001};  // LIH
   RomParaulaControl[10] = {18'b100001001000010};  //BNEG && BZ
   RomParaulaControl[11] = {18'b000000000000000};  //---
   RomParaulaControl[12] = {18'b100000010000011};  //JR
   RomParaulaControl[13] = {18'b100000010000010};  //JMP
   RomParaulaControl[14] = {18'b100010110000000};  //JMPI
   RomParaulaControl[15] = {18'b110000000000001};  //SUB
end
```

FIGURA 245 – Exercici 11 solució a). Modificacions a la rom paraula de control, per incorporar la fusió de les instruccions BNEG i BZ.

També cal modificar les assignacions a les senyals del mateix fitxer UC.v, ja que s'ha modificat la paraula de control:

```
assign paraulaControl =  RomParaulaControl[Instruccio[15:12]];
    assign RA = Instruccio[11:8];
    assign RD = Instruccio[7:4];
    assign RB = Instruccio[3:0];

    assign bz = paraulaControl[6] & Instruccio[7];
    assign bneg = paraulaControl[6] & ~Instruccio[7];

    assign op = paraulaControl[15:13];
    assign alu_rotr = paraulaControl[12];
    assign b_off = paraulaControl[11];
    assign off4_off8 = paraulaControl[10];
    assign off12 = paraulaControl[9];
    assign jmp = paraulaControl[8];
    assign wrMemData = paraulaControl[5];
    assign up_mem = paraulaControl[4];
    assign RAenRD= paraulaControl[3];
    assign RDenRB = paraulaControl[2];
    assign RAenRB = paraulaControl[1];
    assign w_Rd = paraulaControl[0]  & nova_inst;

    assign nova_inst = (exec & ~ld_st) | (mem & ld_st);
```

FIGURA 246 – Exercici II solució a). Modificacions UC, per incorporar la fusió de les instruccions BNEG i BZ.

Finalment per tractar l'immediat s'ha afegit un multiplexor tal i com mostra el següent codi:

```
wire [15:0] ooffset;
mux2_1_16bit MUX_B_COND({8'b00000000,Instruccio[6],Instruccio[6:0]},
                                    Instruccio, bz|bneg, ooffset);
/****************** ENTRADA OFFSET ************************************/
OffsetIn IN_OFFSET (ooffset,off4_off8,off12,OutOffset);
/****************** FI ENTRADA OFFSET *********************************/
```

FIGURA 247 – Exercici II solució a). Modificacions generació de l'Immediat per incorporar la fusió de les instruccions BNEG i BZ.

b) En el cas mostrat anteriorment no fa falta modificar la Unitat de Procés.

c) Veure apartat: *Recursos per Exercicis – Exercici11 – c).*

d)

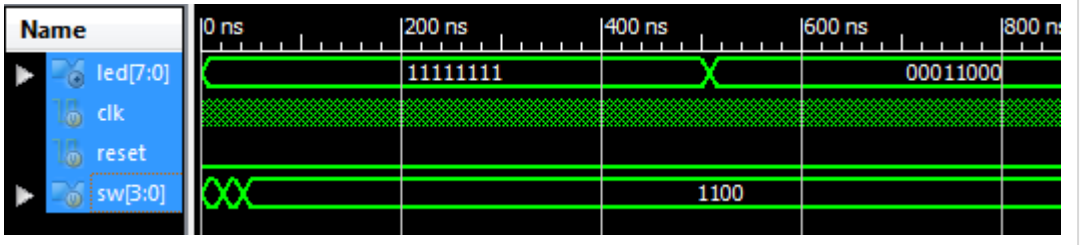


FIGURA 248 – Exercici II solució d). Cronograma que mostra el correcte funcionament del programa Factorial amb les modificacions realitzades al processador.

EXERCICI 12: INSTRUCCIÓ CMP

ENUNCIAT: Aprofitant el codi d'operació 11, que ha quedat lliure al implementar la fusió de les instruccions de salt condicional, amplia el processador *CAL16* per a que el seu conjunt d'instruccions inclogui una nova instrucció de comparació de nombres:

```
CMP Rd, Ra, Rb:   1011 aaaa ddd bbb
```

FIGURA 249 – Exercici 12 enunciat. Especificació de la nova instrucció CMP:

Aquesta es comportarà de la següent manera:

```
Si Ra == Rb        Rd[0] = 1
Si Ra < Rb (NATURALS)  Rd[1] = 1
Si Ra < Rb (ENTERS)    Rd[15] = 1
```

FIGURA 250 – Exercici 12 enunciat. Pseudocodi que especifica el comportament de la instrucció CMP:

Segueix els següents passos:

a) Implementa en Verilog un bloc que realitzi la comparació de dos nombres de 16 bits d'entrada i de sortida tingui un bit associat a cada tipus de comparació definida anteriorment. Crea un fitxer de test i comprova el seu correcte funcionament.

b) Afegeix el bloc anterior a la Unitat de Procés del CAL 16, concretament dins del bloc ALU.

c) Modifica la paraula de control generada per aquesta instrucció en el fitxer de UC, per que aquesta serveixi les senyals de control necessàries.

d) Modifica els fitxers del compilador amb els que se't donen a l'apartat: *Recursos per Exercicis – Exercici12 – d).* Compila el compilador amb la comanada make.

e) Programa una funció auxiliar anomenada **màxim**, la qual donats dos nombres enters que arriben per R1 i R2 respectivament, retorna el màxim en R0. Fes una programa principal que l'utilitzi per comprovar el correcte funcionament. Compara el teu codi de la subrutina, amb el codi de la subrutina funció màxim de l'apartat de integració i proves de la memòria que encara no disposava de la instrucció.

SOLUCIÓ:

a) A continuació es mostra una possible implementació del bloc CMP i del seu fitxer de test:

```
module Comparador(A,B,W);
    input [15:0] A,B;
    output [15:0] W;

    wire [15:0] W_sub;
    wire b,ovf,z,W_xor;

    SUB_16b SUB (A,B,W_sub,b,ovf);
    DetectorZero_16bit DET_Z (W_sub,z);
    xor_1b XOR (W_sub[15],ovf,W_xor);

    assign W = {W_xor,13'b0000000000000,b,z};
endmodule
```

FIGURA 251 – Exercici 12 solució a). Codi en Verilog que implementa el bloc CMP:

```
module testCMP;
    reg [15:0] A,B;
    wire [15:0] W;

    Comparador CMP (A,B,W);
    initial begin
        A = 10;
        B = 10;
        #10 A = -4;
        B = 2;
        #10 A = -8;
        B = -2;
        #10 A = 4;
        B = 2;
        #10 A = 4;
        B = -9;
    end
endmodule
```

FIGURA 252 – Exercici 12 solució a). Fitxer de test pel bloc CMP:

El primer cronograma mostra els valors a comparar tractats com a naturals i el segon com a enters:

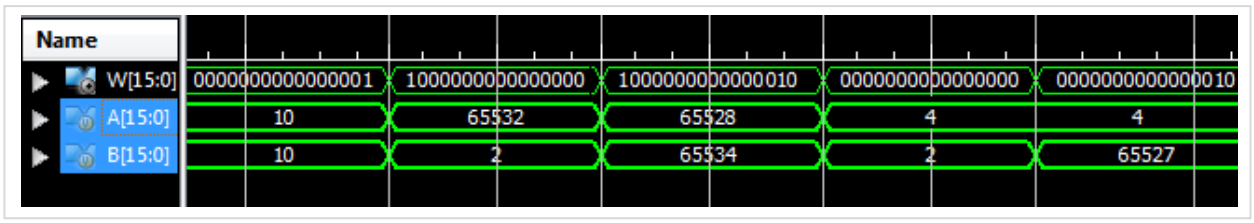


FIGURA 253 – Exercici 12 solució a). Cronograma que mostra el correcte funcionament del bloc CMP per naturals:

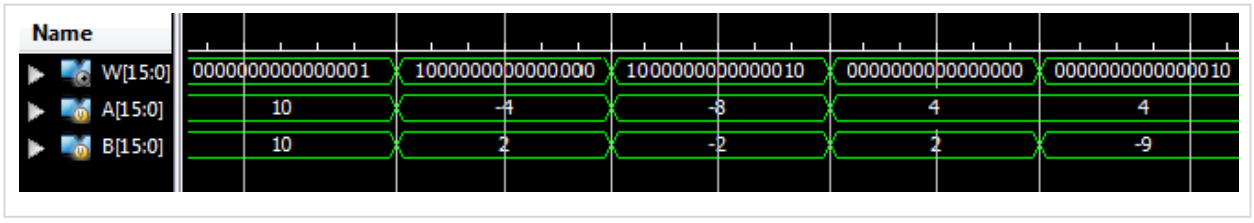


FIGURA 254 – Exercici 12 solució a). Cronograma que mostra el correcte funcionament del bloc CMP per enters.

b)

```
module ALU(A,B,op,Cin,W,z,neg);
    input [15:0] A,B;
    input[2:0] op;
    input Cin;
    output [15:0] W;
    output z,neg;

    wire [15:0] W_and, W_or, W_xor, W_add, W_muxComb, W_sub, W_cmp;
    wire Cout,b,ovf;

    AND_16b AND (A,B,W_and);
    OR_16b OR (A,B,W_or);
    XOR_16b XOR (A,B,W_xor);
    ADD_16b ADD (A,B,Cin,W_add,Cout);
    SUB_16b SUB (A,B,W_sub,b,ovf);
    Comparador CMP (A,B,W_cmp);

    mux8_1_16bit MUX_BLOCS_COMBINACIONALS (W_cmp, W_sub, {B[7:0],A[7:0]},
                                           B, W_add, W_xor, W_or, W_and, op, W);

    assign neg = W[15];
    DetectorZero_16bit DET_Z (W,z);
endmodule
```

FIGURA 255 – Exercici 12 solució b). Codi en Verilog que incorpora el bloc CMP dins el bloc ALU del processador.

c)

```
reg [15:0] RomParaulaControl [15:0];
initial begin
    //op,op,op,alu_rotr,b_off,off4_off8,off12,jmp,b_cond,
    //wrMem,up_mem,RAenRD,RdenRB,RAenRB,wRd
    RomParaulaControl[0] =  {18'b000000000000001};  //AND
    RomParaulaControl[1] =  {18'b001000000000001};  //OR
    RomParaulaControl[2] =  {18'b010000000000001};  //XOR
    RomParaulaControl[3] =  {18'b011000000000001};  //ADD
    RomParaulaControl[4] =  {18'b011010000000001};  //ADDI
    RomParaulaControl[5] =  {18'b000100000000001};  //ROTR
    RomParaulaControl[6] =  {18'b011010000010001};  //LD
    RomParaulaControl[7] =  {18'b011010000100100};  //ST
    RomParaulaControl[8] =  {18'b100011000001001};  //LI
    RomParaulaControl[9] =  {18'b101011000001001};  // LIH
    RomParaulaControl[10] = {18'b100001001000010};  //BNEG && BZ
    RomParaulaControl[11] = {18'b111000000000001};  //CMP
    RomParaulaControl[12] = {18'b100000010000011};  //JR
    RomParaulaControl[13] = {18'b100000010000010};  //JMP
    RomParaulaControl[14] = {18'b100010110000000};  //JMPI
    RomParaulaControl[15] = {18'b110000000000001};  //SUB
end
```

FIGURA 256 – Exercici 12 solució c). Modificació rom paraula de control per incorporar la nova instrucció CMP.

d) Veure apartat: *Recursos per Exercicis – Exercici12 – d).*

e)

```
etiq_maxim:
        CMP R0,R1,R2
        BNEG R0, etiq_else
        ADDI R0,R1,0
        JMP R15
etiq_else:
        ADDI R0,R2,0
        JMP R15
```

FIGURA 257 – Exercici 12 solució e). Programa en CAL_assembler que fa ús de la nova instrucció CMP.

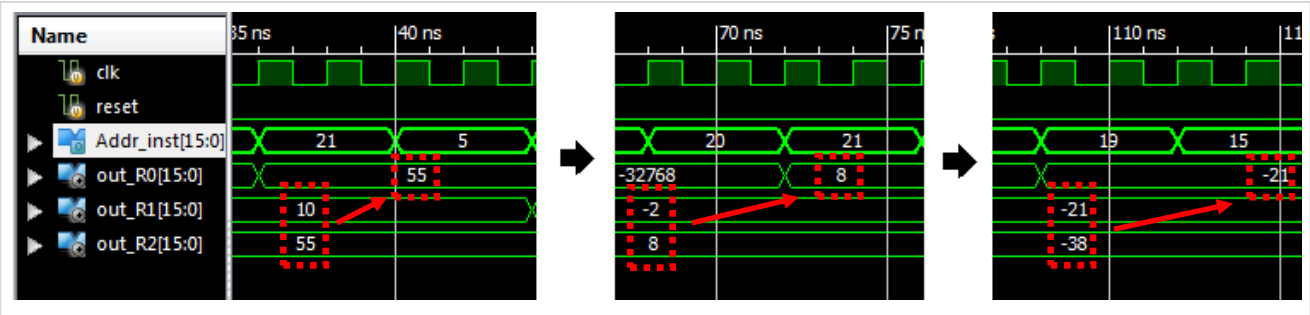


FIGURA 258 – Exercici 12 solució e). Cronograma que mostra el correcte funcionament del programa en CAL_assembler que fa ús de la instrucció CMP.

RECURSOS PER EXERCICIS

EXERCICI 4: BUBBLE SORT

f) En el CD del curs trobaràs els fitxers **ex4b-data.bin i ex4b-program.bin**, seleccionar el seu contingut i copiar-lo dins el fitxer **Processador\Programes\data.bin i Processador\Programes\program.bin** del teu processador, respectivament.

g) En el CD del curs trobaràs els fitxers **ex4f-data.bin i ex4f-program.bin**, seleccionar el seu contingut i copiar-lo dins el fitxer **Processador\Programes\data.bin i Processador\Programes\program.bin** del teu processador, respectivament.

EXERCICI 5: FUNCIÓ MULTIPLICACIÓ

b) En el CD del curs trobaràs els fitxers **ex5b-data.bin i ex5b-program.bin**, seleccionar el seu contingut i copiar-lo dins el fitxer **Processador\Programes\data.bin i Processador\Programes\programa.bin** del teu processador, respectivament.

f) En el CD del curs trobaràs el fitxer **ex5f-program.bin**, seleccionar el seu contingut i copiar-lo dins el fitxer **Processador\Programes\program.bin** del teu processador.

EXERCICI 6: FACTORIAL D'UN NOMBRE

b) En el CD del curs trobaràs el fitxer **ex6b-program.bin**, seleccionar el seu contingut i copiar-lo dins el fitxer **Processador\Programes\program.bin** del teu processador.

EXERCICI 7: COMPTADOR LEDS

a) En el CD del curs trobaràs el fitxer **ex7a-program.bin**, seleccionar el seu contingut i copiar-lo dins el fitxer **Processador\Programes\program.bin** del teu processador.

b) En el CD del curs trobaràs el fitxer **white-board.ucf**. Cal crear un fitxer de configuració tal i com explica el **Manual d'Usuari.** Un cop creat copiar el contingut del fitxer **white-board.ucf** al nou fitxer creat i descomentar els dispositius que es necessitin en cada cas.

d) En el CD del curs trobaràs el fitxer **ex7d-video.mov**. Aquest conté una gravació de l'execució del programa anterior amb la FPGA Spartan 3E.

EXERCICI 8: MOSTRA VALORS D'UN VECTOR

a) En el CD del curs trobaràs el fitxer **ex8a-program.bin**, seleccionar el seu contingut i copiar-lo dins el fitxer **Processador\Programes\program.bin** del teu processador.

b) En el CD del curs trobaràs el fitxer **white-board.ucf**. Cal crear un fitxer de configuració tal i com explica el **Manual d'Usuari.** Un cop creat copiar el contingut del fitxer **white-board.ucf** al nou fitxer creat i descomentar els dispositius que es necessitin en cada cas.

d) En el CD del curs trobaràs el fitxer **ex8d-video.avi**. Aquest conté una gravació de l'execució del programa anterior amb la FPGA Spartan 3E.

EXERCICI 9: CONÈIXER L'ESTAT D'UN PROGRAMA

b) En el CD del curs trobaràs el fitxer **white-board.ucf**. Cal crear un fitxer de configuració tal i com explica el **Manual d'Usuari.** Un cop creat copiar el contingut del fitxer **white-board.ucf** al nou fitxer creat i descomentar els dispositius que es necessitin en cada cas.

d) En el CD del curs trobaràs el fitxer **ex9d-video.avi**. Aquest conté una gravació de l'execució del programa anterior amb la FPGA Spartan 3E.

g) En el CD del curs trobaràs el fitxer **ex9g-video.avi**. Aquest conté una gravació de l'execució del programa anterior amb la FPGA Spartan 3E.

EXERCICI 10: INSTRUCCIÓ SUB

d) En el CD del curs trobaràs els fitxers **ex10d-semantic.cc** i **ex10d-codegen.cc** cal que el copiïs al directori **Compilador** amb el nom de **semantic.cc** i **codegen.cc** respectivament.

EXERCICI 11: FUSIÓ BNEG & BZ

c) En el CD del curs trobaràs el fitxer **ex11c-codegen.cc** cal que el copiïs al directori **Compilador** amb el nom de **codegen.cc**.

EXERCICI 12: INSTRUCCIÓ CMP

d) En el CD del curs trobaràs els fitxers **ex12d-semantic.cc** i **ex12d-codegen.cc** cal que el copiïs al directori **Compilador** amb el nom de **semantic.cc** i **codegen.cc** respectivament.

ANNEXES

ANNEX 1 – *SCRUM*

SCRUM és un metodologia àgil que es porta a terme en l'àmbit de la gestió i desenvolupament de projectes. És caracteritza per estar basat en un procés iteratiu i incremental. Tot i estar enfocat a la gestió de processos de desenvolupament de software, aquest té múltiples aplicacions en altres tipus de projectes.

Scrum és un model de referències que defineix un conjunt de pràctiques i rols com a punt de partida. Els rols principals són ScrumMaster que pren el paper de director del projecte, Scrum Owner que representa el client i el Team, l'equip de treball.

Amb SCRUM el temps s'organitza en sprints, un període limitat i no molt llarg de temps en el que es desenvolupa una part del projecte que es pot entendre com a potencialment entregable. El conjunt de característiques de cada sprint ve definit pel Product Backlog, que llista i especifica les tasques a realitzar, i aquest es defineix durant les reunions, Sprint Planning. Durant les reunions el Scrum Owner identifica els elements i desenvolupa el Product Backlog i llavors l'equip avalua si la càrrega de feina és factible. Durant un sprint no està permès canviar els requisits definits al Product Backlog.

ANNEX 2 – *Complement a dos*

Forma de representació de nombres enters en binar. El càlcul del complement a dos és molt senzill i molt útil ja que es pot realitzar amb portes lògiques. La seva utilitat principal és per realitzar operacions matemàtiques amb nombres binaris, en particular, facilitat molt el càlcul de la resta binària.

La representació dels nombres positius queda exactament igual que la representació binaria del nombre en concret, però sempre garantint que el bit de més pes val 0, que és el que identifica els nombres positius. Per representar nombres negatius, cal agafar el valor absolut del nombre en binari a representar, negar bit a bit el valor de les seves xifres, és a dir, realitzar el complement a un, i finalment incrementar en una unitat el nombre. Aquests es caracteritzen per tenir el primer dígit sempre a 1, el que indica que és un nombre negatiu.

Càlcul de –X: (on X és un natural codificat en binari amb n bits)

$$-X = \sim |X| + 1$$

Exemple: Calcular la codificació binària en Ca2 del nombre –54 amb 8 bits.

Valor absolut en binari:	54 = 00110110
Càlcul del Ca2:	~00110110 = 11001001
	11001001 + 1 = 11001010 = -54

FIGURA 259 – Fórmula del càlcul del Complement a dos d'un nombre binari.

Per representar el signe, s'utilitza el bit de més pes, tal i com ja s'ha comentat, un 0 indica un nombre positiu i un 1 indica un nombre negatiu. Aquest fet implica la pèrdua de rang de representació respecta a la codificació normal. Per a calcular el rang de representació dels nombres enters codificats en n bits, s'utilitza la següent fórmula:

Fórmula del Rang: $-2^{n-1} \leq Rang \leq 2^{n-1} - 1$ on n és el nombre de bits de representació.

Exemple: Calcular el rang de representació en Ca2 per nombres de 8 bits.

Rang = $[-2^{n-1}, +2^{n-1} - 1] = [-2^{8-1}, +2^{8-1} - 1] =$ **$[-128, +127]$**

FIGURA 260 – Fórmula del rang dels nombres enters codificats en Complement a dos.

ANNEX 3 – *CD Adjunt*

El CD adjunt en aquest apartat conté diferents recursos relacionats amb el projecte. Concretament la seva estructura és la següent:

CAL16

Documents **Memòria** → Memòria del Projecte en format .pdf

Planificació → Fitxer de planificació en format .mp

Presentació → Presentació oral en format .pptx

Processador Projecte que inclou el codi font del processador en Verilog.

Compilador Fitxers que formen el compilador de CAL_assembler.

Demos Vídeos de demostració dels exercicis del capítol Memòria Tècnica, apartat Integració i Proves

Recursos Exercicis Fitxers necessaris pel Document Adjunt,Col·lecció d'Exercicis.

FIGURA 61 – CD de recursos auxiliars a la Memòria

ANNEX 4 – *Instruccions del CAL_assembler*

Per a cada instrucció s'ha definit la seva estructura en assemblador i en binari, la seva semàntica, se'n fa una breu descripció i es dóna un exemple de funcionament. Observem que tal i com s'ha descrit en la Memòria, el *CAL16* és un processador que implementa seqüenciament implícit, i per aquesta raó totes aquelles instruccions que no realitzen una ruptura de seqüència la seva semàntica inclou un increment en l'estat actual del acumulador de programa (PC).

AND

Binari: 0000 aaaa dddd bbbb

Assemblador: AND Ra, Rd, Rb

Semàntica: Rd = Ra & Rb
PC = PC + 1

Descripció: Aquesta operació realitza una AND bit a bit entre els bits de mateix pes que es troben emmagatzemats en els registres Ra i Rb, la tira de bits resultat es guarda al registre destí Rd.

Exemple: AND R1, R2, R3 on R2=0x16A5 i R3=0x5AB6

```
  0001 0110 1010 0101
& 0101 1010 1011 0110
  0001 0010 1010 0100   →   R1 = 0x12A4
```

OR

Binari: 0001 aaaa dddd bbbb

Assemblador: OR Ra, Rd, Rb

Semàntica: Rd = Ra | Rb
PC = PC + 1

Descripció: Aquesta operació realitza una OR bit a bit entre els bits de mateix pes que es troben emmagatzemats en els registres Ra i Rb, la tira de bits resultat es guarda al registre destí Rd.

Exemple: OR R1, R2, R3 on R2=0x16A5 i R3=0x5AB6

```
  0001 0110 1010 0101
| 0101 1010 1011 0110
  0101 1110 1011 0111   →   R1 = 0x5EB7
```

XOR

Binari: 0010 aaaa dddd bbbb

Assemblador: XOR Ra, Rd, Rb

Semàntica: Rd = Ra ^ Rb
PC = PC + 1

XOR

Descripció: Aquesta operació realitza una XOR bit a bit entre els bits de mateix pes que es troben emmagatzemats en els registres Ra i Rb, el resultat es guarda al registre destí Rd.

Exemple: XOR R1, R2, R3 on R2=0x16A5 i R3=0x5AB6

```
  0001 0110 1010 0101
^ 0101 1010 1011 0110
  0100 1100 0001 0011   →   R1 = 0x4C13
```

ADD

Binari: 0011 aaaa dddd bbbb

Assemblador: ADD Ra, Rd, Rb

Semàntica:
Rd = Ra + Rb
PC = PC + 1

Descripció: Aquesta operació realitza una suma entre els nombres que es troben emmagatzemats en els registres Ra i Rb, guardat el resultat al registre destí Rd. Els nombres es tracten com a enters de 16 bits codificats en Ca2. En cap cas es detecta si el resultat és representable, sinó que en cas que no ho sigui, el resultat que es donarà serà aquell representable amb 16 bits, és a dir es trunca[5].

Exemple: ADD R1, R2, R3 on R2=0x16A5 i R3=0x5AB6

```
  0001 0110 1010 0101
+ 0101 1010 1011 0110
  0111 0001 0101 1011   →   R1 = 0x715B
```

ADDI

Binari: 0100 aaaa dddd oooo

Assemblador: ADDI Ra, Rd, Offset-4b

Semàntica:
Rd = Ra + Sx(Offset-4b)[1]
PC = PC + 1

Descripció: Aquesta operació realitza una suma del nombre que es troba emmagatzemat en el registre Ra més el l'immediat representat amb 4 bits Offset-4b, un nombre amb rang entre [-8 , 7], El resultat es guarda al registre destí Rd. Els nombres es tracten com a enters de 16 bits codificats en Ca2. Per a l'offset s'agafen els 4 bits de menys pes i se li estén el signe fins aconseguir un enter de 16 bits. En cap cas es detecta si el resultat és representable, sinó que en cas que no ho sigui, el resultat que es donarà serà aquell representable amb 16 bits, és a dir, els bits de més pes no representables es perden.

Exemple: ADDI R1, R2, -5 on R2=0x16A5

```
  0001 0110 1010 0101
+ 1111 1111 1111 1011
  0001 0110 1010 0000   →   R1 = 0x16A0
```

[1] *Sx(X) : Nomenclatura utilitzada com a funció X. Indica que el valor X (del propi immediat) se li estén el signe fins a obtenir un enter codificat amb 16 bits.*

ROTR

Binari: 0101 aaaa dddd oooo

Assemblador: ROTR Ra, Rd, Offset-4b

Semàntica: Rd = Ra rot Offset-4b
PC = PC + 1

Descripció: Aquesta operació realitza una rotació del nombre que es troba emmagatzemat en el registre Ra, guardant el nombre rotat en el registre destí Rd. L'immediat es tracta com a un nombre natural de 4 bits. El sentit de la rotació és cap a l'esquerra. Es desplacen els bits emmagatzemats en Ra tantes posicions com el nombre que val l'Offset, el que significa que podem portar a terme rotacions dins el rang [0 , 15]. Els bits de més pes passen a ser els de menys pes a mesura que es surten de rang en els desplaçaments unitaris.

Exemple: ROTR R1, R2, 5 on R2=0x16A5

0001 0110 1010 0101
rot 0x5
1101 0100 1010 0010 → R1 = 0xD4A2

LD

Binari: 0110 aaaa dddd oooo

Assemblador: LD Rd, Offset-4b(Ra)
Semàntica: Rd = Mem [Ra + Sx(Offset-4b)]
PC = PC + 1

Descripció: Aquesta operació realitza un accés a la posició de la Memòria de Dades [7] resultant de sumar el contingut del registre Ra més el valor de Offset-4b. L'immediat es tracta com un nombre enter de 4 bits i se li estén el signe fins aconseguir un enter de 16 bits, el rang d'aquest és entre [-8 , 7]. Un cop obtingut el valor de la posició concreta de Memòria de Dades aquest s'emmagatzema en el registre destí Rd.

Exemple: LD R1, -3(R2) on R2=0x000A i MemData[7]=0xFFFF

R1 = MemData[0x000A + 0xFFFD] = MemData[0x0007] = 0xFFFF
→ R1 = 0xFFFF

ST

Binari: 0111 aaaa dddd oooo

Assemblador: ST Offset-4b(Ra), Rb

Semàntica: Mem [Ra + Sx(Offset-4b)] = Rb
PC = PC + 1

ST

Descripció: Aquesta operació realitza una escriptura del valor que es troba en Rb, a la posició de la Memòria de Dades resultant de sumar el contingut del registre Ra més el valor de Offset-4b. L'immediat es tracta com un nombre enter de 4 bits i se li estén el signe fins aconseguir un enter de 16 bits, el rang d'aquest és entre [-8 , 7].

Exemple: ST 6(R1), R2 on R1=0x0005 i R2=0x16A5

MemData[0x0005 + 0x0006] = MemData[0x000B] = R2 = 0x16A5

→ MemData[0x000B] = 0x16A5

LI

Binari: 1000 aaaa oooooooo

Assemblador: LI Rd, Io (Offset-16b)

Semàntica:
Rd = Sx(Offset-8b)
PC = PC + 1

Descripció: Aquesta operació permet la càrrega d'immediats a registres. L'immediat es tracta com un nombre enter de 8 bits (els de menys pes) i se li estén el signe fins aconseguir un enter de 16 bits, el rang d'aquest és entre [-128 , 127].

Exemple: LI R1, -1 → R1 = 0xFFFF

LIH

Binari: 1001 aaaa oooooooo

Assemblador: LIH Rd, hi (Offset-16b)

Semàntica:
Rd = (Rd & 0x00FF) | Offset-8b
PC = PC + 1

Descripció: Aquesta operació permet la càrrega d'immediats a registres. S'agafen els 8 bits de l'immediat i es col·loquen en els 8 bits de més pes de Rd, deixant els 8 bits de menys pes amb el mateix valor.

Exemple: LIH R1, -1 on R1=0x0005 → R1 = 0xFF05

BNEG

Binari: 1010 aaaa oooooooo

Assemblador: BNEG Ra, Offset-8b

Semàntica:
si (Ra < 0)
  PC = PC + Sx(Offset-8b)
sino
  PC = PC + 1
fsi

Descripció: Aquesta és una operació de ruptura de seqüència condicional i relativa el PC. La instrucció avalua el valor que conté el registre Ra tractat com a enter codificat en Ca2 amb 16 bits, en cas que aquest sigui negatiu, s'incrementa l'estat del comptador de pro-

BNEG

Descripció: grama amb el valor de l'immediat. L'immediat es tracta com un nombre enter codificat en Ca2 de 8 bits i se li estén el signe fins aconseguir un enter de 16 bits, el rang d'aquest és entre [-128 , 127].

Exemple:

BNEG R1, -2 — on R1=0x0005 i PC = 0x000A
PC = PC + 1 = 0x000A + 0x0001 = 0x000B → PC = 0x000B

BNEG R1, -2 — on R1=0xFFFA i PC = 0x000A
PC = PC + (-2) = 0x000A + 0xFFFE = 0x0008 → PC = 0x0008

BZ

Binari: 1011 aaaa oooooooo

Assemblador: BZ Ra, Offset-8b

Semàntica:

```
si ( Ra == 0 )
  PC =  PC + Sx( Offset-8b )
sino
  PC =  PC + 1
fsi
```

Descripció: Aquesta és una operació de ruptura de seqüència condicional i relativa el PC. La instrucció avalua el valor que conté el registre Ra tractat com a enter codificat en Ca2 amb 16 bits, en cas que aquest sigui igual a 0, s'incrementa l'estat del comptador de programa amb el valor de l'immediat. L'immediat es tracta com un nombre enter codificat en Ca2 de 8 bits i se li estén el signe fins aconseguir un enter de 16 bits, el rang d'aquest és entre [-128 , 127].

Exemple:

BZ R1, -2 — on R1=0x0005 i PC = 0x000A
PC = PC + 1 = 0x000A + 0x0001 = 0x000B → PC = 0x000B

BZ R1, -2 — on R1=0x0000 i PC = 0x000A
PC = PC + (-2) = 0x000A + 0xFFFE = 0x0008 → PC = 0x0008

JZ

Binari: 1100 aaaa dddd oooo

Assemblador: JZ Ra, Rd, Offset-4b

Semàntica:

```
Rd = PC
PC =  Ra + Sx( Offset-4b )
```

Descripció: Aquesta és una operació de ruptura de seqüència absoluta i que crea una link a l'estat actual del PC. La instrucció realitza les dues assignacions de forma paral·lela. Per una banda el comptador de programa PC pren el valor resultant de la suma del nombre que conté el registre Ra, tractat com a enter codificat en Ca2 amb 16 bits, més el valor de l'immediat. L'immediat es tracta com un nombre enter codificat en Ca2 de 4 bits i se li estén el signe fins aconseguir un enter de 16 bits, el rang d'aquest és entre [-8 , 7]. I per l'altre, emmagatzema el valor de PC (abans que prengui el resultat de la suma) en el registre destí Rd.

Exemple:

JZ R1, R2, -2 — on R2=0x0005 i PC = 0x000A

PC = R2 + (-2) = 0x0005 + 0xFFFE = 0x0003 → PC = 0x0003
R1 = PC = 0x000A → R1 = 0x000A

JMP

Binari: 1101 aaaa 00000000

Assemblador: JMP Ra

Semàntica: PC = Ra

Descripció: Aquesta és una operació de ruptura de seqüència absoluta. La instrucció realitza l'assignació al registre del PC el valor del registre Ra, Els últims 8 bits de la instrucció valen 0, ja que és així com ho genera el compilador. La direcció destí del salt trobarà codificada en els 12 bits de menys pes de Ra.

Exemple: JMP R2 on R2=0050 → PC = 0802

JMPI

Binari: 1110 oooooooooooo

Assemblador: JMPI Offset-12b

Semàntica: PC = Sx(Offset-12b)

Descripció: Aquesta és una operació de ruptura de seqüència absoluta. La instrucció realitza l'assignació al registre del PC el valor de l'immediat. L'immediat es tracta com un nombre natural codificat amb 12 bits, ja que aquesta és la mida màxima per a direccionar la Memòria d'Instruccions.

Exemple: JMP 2050 2050 = 0x802 (en 12 bits)

PC = 0xF802 & 0x0FFF = 0x0802 → PC = 0802

ANNEX 5 – *Arbre de Sintaxis Abstracte (AST)*

Estructura de dades en forma d'arbre utilitzat en ciències de la computació per a representar un codi font escrit en un llenguatge. Cada node de l'arbre representa una construcció del codi font, i l'herència de l'arbre simbolitza la pertinència de les construccions en d'altres construccions més globals.

La sintaxis és abstracta ja que no representa tota la informació que apareix en el codi, si no que tant sols emmagatzema aquella necessària, juntament amb informació relativa per a realitzar comprovacions de tipus semàntic durant un procés de compilació.

ANNEX 6 – ANTLR

Una altra eina pel reconeixement de llenguatge, és una eina desenvolupada per Terence Parr, basada en el projecte de PCCTS de la Universitat de Purdue. És un projecte sota la llicència lliure de BSD (Berkeley Software Distribution), que permet disposar del codi font i es pot executar sota les plataformes de Linux, Windows i Mac OS X.

ANTLR és un eina que opera sobre el llenguatges, proporcionant un marc de construcció de parsers, intèrprets, compiladors i traductors de llenguatges a partir de la seva descripció gramatical.

Es considera un meta-programa, ja que crea nous programes, partint de la descripció formal d'una gramàtica d'un llenguatge, ANTLR genera un programa que determina si una sentència pertany o no al llenguatge utilitzant algorismes LL(*) de parsing. A més proporciona facilitat per a crear estructures intermitges a anàlisis, com per exemple ASTs, per recorre'ls i proveeix mecanismes d'autorecuperació i identificació d'errors.